Performance Analysis using Coloured Petri Nets

Lisa Wells

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Performance Analysis using Coloured Petri Nets

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Lisa Wells
July 31, 2002

Abstract

Performance is often a central issue in the design, development, and configuration of systems. It is not always enough to know that systems work properly, they must also work effectively. There are numerous studies, e.g. in the areas of computer and telecommunication systems, manufacturing, military, health care, and transportation, that have shown that time, money, and even lives can be saved if the performance of a system is improved. Performance analysis studies are conducted to evaluate existing or planned systems, to compare alternative configurations, or to find an optimal configuration of a system. There are three alternative techniques for analysing the performance of a system: measurement, analytical models, and simulation models.

This dissertation focuses on the the use of coloured Petri nets for simulation-based performance analysis of industrial-sized systems. Coloured Petri nets are particularly well suited for modelling and analysing large and complex systems for several reasons: they have an intuitive graphical representation; they are executable; hierarchical models can be constructed; it is possible to model the time used by different activities in a system; and mature and well-tested tools exist for creating, simulating, and analysing coloured Petri net models.

The dissertation consists of two parts. Part II is composed of four individual papers and constitutes the core of the dissertation. All four papers have been published or accepted for publication as workshop papers. Part I is the obligatory overview paper with summarises the work that has been done. The overview paper introduces the research field of performance analysis using coloured Petri nets, and it summarises the contents and contributions of the four individual papers.

The first paper presents an overview of improved facilities for performance analysis using coloured Petri nets. Personal experience has shown that people with different backgrounds have very different needs with regards to tools supporting simulation-based performance analysis. Inexperienced data analysts will have a tendency to believe what a tool tells them, therefore care must be taken to avoid generating misleading results. More experienced data analysts generally require that more sophisticated kinds of data are generated for specific purposes. The paper presents new performance-related facilities such as support for running multiple simulations, calculating confidence intervals, generating organised and systematic simulation output, simulating and comparing alternative system configurations, and variance reduction techniques.

The second paper presents a framework for implementing monitoring facilities that observe, inspect, and control simulations of coloured Petri nets. During

the past decade, a number of libraries and tool extensions have been developed for tools supporting coloured Petri nets, including support for data collection, message sequence charts, updating domain-specific graphics, and communication between simulators and other processes. While there are many advantages to having this extra functionality, there are a number of disadvantages as well. The most serious problem is that it is often necessary to modify the behaviour of a model in order to use the libraries and tool extensions. With this framework it becomes possible to make an explicit separation between modelling the behaviour of a system and monitoring the behaviour of the model. As a result, cleaner and more understandable models can be created.

The third paper presents a novel method for adding auxiliary information to coloured Petri net models. Coloured Petri nets models can be used for several fundamentally different purposes such as functional analysis, visualisation, and performance analysis. It is seldom the case that the exact same model can be used for a variety of different purposes, as it is frequently necessary to make small or large modifications to the model in order to obtain a model that is appropriate for another purpose. There are a number of disadvantages associated with modifying a model. Making the modifications can be time consuming and error-prone. It is tiresome to maintain several different versions of a model during development. More importantly, there is no guarantee that the behaviour of the model will not be affected by the modifications that are made. The main advantages of the proposed method are: auxiliary information can be defined separately from a model, the auxiliary information is shown to affect the behaviour of a model in a very limited and predictable manner, and it is easy to enable and disable the auxiliary information.

The fourth paper is a case study in which the performance of a web server was analysed using coloured Petri nets. This case study has shown that it is relatively easy to analyse the performance of an industrial-sized system using coloured Petri nets and the improved performance facilities that are described in the first paper. The case study demonstrated that typical users of coloured Petri nets are not experienced performance analysts, and that this fact ought to be taken into consideration when developing performance-related facilities for coloured Petri net tools.

Acknowledgements

There are many people who deserve to be thanked for their part in helping me get to where I am today. I can only mention a few here.

I would like to start by thanking Søren Christensen and Kurt Jensen for providing invaluable supervision, guidance, and inspiration during the past several years. They hired me as a student programmer, supervised my Master's thesis, encouraged me to apply to the PhD program, and then supervised my PhD studies. For this, I am extremely grateful because it has given me ample opportunity to broaden my horizons and to challenge myself.

I have been very fortunate to be a part of the CPN Group at the University of Aarhus during the past four and a half years. Being a member of this group has been both scientifically rewarding and personally enriching. Several members of the group deserve special recognition. Thanks to Bo Lindstrøm and Louise Elgaard with whom I have shared the joys and tribulations of being a PhD candidate. I am very pleased to have had the opportunity to work closely with Bo. After writing our Master's thesis together, our research interests diverged when we started our PhD studies. Our interests converged again towards the end of our studies, which has resulted in two papers. Thanks to Lars M. Kristensen who has taken the time to act as a mentor for the next generation of PhD students within the CPN Group. Thomas Mailund, Kjeld H. Mortensen, and Jens Bæk Jørgensen must also be thanked for providing thorough and thoughtful criticism of several papers during the past years.

I would also like to thank: Søren Glud Johansen, of the Operations Research Department at the University of Aarhus, for teaching me how to think like a data analyst; to Professor Rajive Bagrodia for hosting an informative 5 month stay with the Parallel Computing Lab at the University of California at Los Angeles, USA; Professor Frederic "Ty" Cunningham Jr., of Bryn Mawr College, USA, for encouraging me to consider pursuing a teaching career; and my parents for continually supporting my decisions, regardless of how far away from home these decisions have led me.

I would especially like to thank Finn for his untiring show of love, patience and support, and for reminding me about what is truly important.

The work presented in this dissertation has been funded by the Danish National Centre for IT Research and the Hewlett-Packard Corporation.

Lisa Wells
Århus, July 31, 2002

Contents

Part I

Overview

Chapter 1

Introduction

performance (*noun*) the manner in which or the efficiency with which something reacts or fulfils its intended purpose.

- Random House Webster's College Dictionary

This chapter introduces the research field of performance analysis using coloured Petri nets. Section 1.1 gives a general introduction to performance analysis. Section 1.2 provides a brief introduction to coloured Petri nets. Section 1.3 gives an introduction to performance analysis using coloured Petri nets. Section 1.4 presents the motivation and aims of this dissertation. Section 1.5 gives an overview of the work done and the structure of this dissertation, and it includes an outline for the remainder of this overview paper.

1.1 Performance Analysis

Performance is often a central issue in the design, development, and configuration of systems. It is not always enough to know that systems work properly, they must also work effectively. There are numerous studies, e.g. in the areas of computer and telecommunication systems, manufacturing, military, health care, and transportation, that have shown that time, money, and even lives can be saved if the performance of a system is improved. Performance analysis studies are conducted to evaluate existing or planned systems, to compare alternative configurations, or to find an optimal configuration of a system. There are three alternative techniques for analysing the performance of a system: measurement, analytical models, and simulation models. There are advantages and drawbacks to each of these techniques.

Measuring the performance of a system can provide exact answers regarding the performance of the system. The system in question is observed directly — no details are abstracted away, and no simplifying assumptions need to be made regarding the behaviour of the system. However, measurement is only an option if the system in question already exists. The measurements that are taken may or may not be accurate depending on the current state of the system. For example, if the utilization of a network is measured during an off-peak period, then no conclusions can be drawn about either the average utilization of the network or the utilization of the network during peak usage periods.

Analytical models, such as Markovian models [60], can provide exact results regarding the performance of a system. The results are exact, in that they are not estimates of the performance of the system. However, the results provided by analytical models may or may not be accurate, depending on the assumptions that have been made in order to create the model. In many cases it is difficult to accurately model industrial-sized systems with analytical models. In fact, Jain [69] has observed that when analysing computer systems "analytical modeling requires so many simplifications and assumptions that if the results turn out to be accurate, even the analysts are surprised."

Simulation-based performance analysis can be used as an alternative to analytical techniques. Simulation can rarely provide exact answers, but it is possible to calculate how precise the estimates are. Furthermore, larger and more complex models can generally be created and analysed without making restrictive assumptions about the system. There are two main drawbacks to using simulation: it may be time consuming to execute the necessary simulations, and it may be difficult to achieve results that are precise enough. Simulation-based performance analysis of a model involves a statistical investigation of output data, the exploration of large data sets, the appropriate visualisation, and the verification and validation of simulation experiments.

Performance analysis is both an art and a science. One of the arts of performance analysis is knowing which of these three analysis technique to use in which situation. Measurement can obviously not be used if the system in question does not exist. Simulation should probably not be used if the system consists of a few servers and queues, in this case queueing networks [78] would be a more appropriate method. Simulation and analytic models are often complementary. Analytic models are excellent for smaller systems that fulfil certain requirements, such as exponentially distributed interarrival periods and processing times. Simulation models are more appropriate for large and complex systems with characteristics that render them intractable for analytic models. Performance analysts need to be familiar with a variety of different techniques, models, formalisms and tools. Creating models that contain an appropriate level of detail is also an art. It is important to include enough information to be able to make a reasonable representation of the system, however, it is equally important to be able to determine which details are irrelevant and unnecessary.

Simulation-based performance analysis is the focus of this dissertation. One of the sciences associated with simulation studies is the application of the appropriate statistical techniques when analysing simulation output. After a model has been created and validated, there are a multitude of decisions that need to be made before a study can proceed. *Experimental design* [84] is concerned with determining which scenarios are going to be simulated and how each of the scenarios will be simulated in a simulation study. For each scenario in a simulation study one needs to decide how many simulations will be run, how long the simulations will be, and how each scenario will be initialised. Both the length of a simulation and the number of replications run can have a significant impact on the performance measure estimates. Making arbitrary, unsystematic or improper decisions is a waste of time, and the simulation study may fail to produce useful results [69].

In some studies, the scenarios may been given, and the purpose of the study may be to compare the performance of the given configurations with a standard or to choose the best of the configurations. If the scenarios are not predetermined, then the purpose of the simulation study may be to locate the parameters that have the most impact on a particular performance measure or to locate important parameters in the system. *Sensitivity analysis* [77] investigates how extreme values of parameters affect performance measures. *Gradient estimation* [84], on the other hand, is used to examine how small changes in numerical parameters affect the performance of the system. *Optimisation* [3] is often just a sophisticated form of comparing alternative configurations, in that it is a systematic method for trying different combinations of parameters in hope of finding the combination that gives the best results.

1.2 Coloured Petri Nets

This dissertation focuses on the the use of *coloured Petri nets* [70, 80] (CP-nets or CPN) for performance analysis. CP-nets are a graphical modelling language that model both the states of a system and the events that change the system from one state to another. CP-nets combine the strengths of *Petri nets* [103] (PN) and programming languages. The formalism of Petri nets is well suited for describing concurrent and synchronising actions in distributed systems. Programming languages can be used to define data types and manipulation of data. In timed CP-nets [71], which are described in more detail below, a global clock models the passage of time. Large and complex models can be built using hierarchical CP-nets in which modules, which are called *pages* in CPN terminology, are related to each other in a well-defined way. Without the hierarchical structuring mechanism, it would be difficult to create understandable CP-nets of real-world systems.

CP-nets can be used in practice only because there are mature and well-tested tools supporting them. The Design/CPN tool [31, 40] was first released in 1989, and it supports editing, simulation, and analysis of CP-nets. CPN Tools [37] was released in October 2001 and will eventually replace Design/CPN. CPN Tools has a new GUI with state-of-the-art interaction techniques [16], such as two-handed input, toolglasses and marking menus, and an improved simulator [96]. The *inscription language*, i.e. the language for defining data types and for modifying data, for both tools is the programming languages is Standard ML [100, 116].

Petri nets provide a framework for modelling and analysing both the performance and the functionality of distributed and concurrent systems. A distinction is generally made between *high-level Petri nets* and *low-level Petri nets*. CP-nets are an example of high-level Petri nets which combine Petri nets and programming languages and which are aimed at modelling and analysing realistically-sized systems.. Low-level Petri nets to have a simpler graphical representation, and they are well suited as a theoretical model for concurrency. The Petri Net World web site [105] contains extensive descriptions of the many kinds of Petri nets and of PN tools.

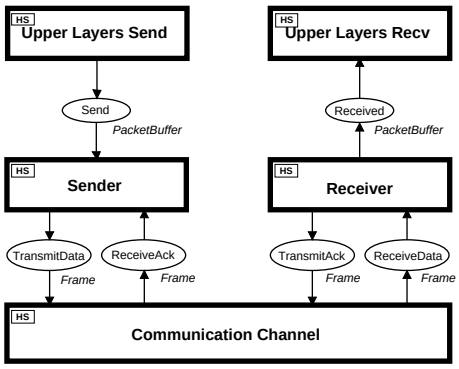


Figure 1.1: Top-level view of CP-net of stop-and-wait protocol.

Example Figure 1.1 shows the most abstract view of a timed, hierarchical CPN model. The model represents a stop-and-wait protocol from the data link control layer of the OSI network architecture. In Part I of this dissertation, the model will be referred to as the *stop-and-wait model*. The details of the model are not discussed here, as the model will only be used to introduce basic concepts related to CP-nets. The model is taken from [80], and it is used as an example in the paper that is discussed in Chapters 2 and 7.

The stop-and-wait protocol provides reliable communication in a system which consists of a sender transmitting data packets to a receiver across an unreliable, bi-directional communication channel. The sender accepts data packets from protocols in the upper layers of the OSI network architecture. Similarly, the receiver passes packets that have been properly received to the upper layers of the protocol stack. The stop-and-wait model consists of a number of pages, and Fig. 1.1 shows the page named SWprotocol. The Upper Layers Send page generates workload for model. The Communication Channel page provides a simple model of an unreliable network in which packet loss and overtaking can occur. The protocol is modelled in detail in the Sender and Receiver parts of the model.

Figure 1.2 shows the Sender part of the stop-and-wait model. The states of a CP-net are represented by a number of *tokens* positioned on *places*, which are drawn as ellipses. No tokens are shown in Fig 1.2. Each token carries a data value. The data value may be simple, such as an integer or a string, or it may be complex, such as a pair consisting of a list of integers and a boolean. The events of a CP-net are represented by means of *transitions*, which are drawn as rectangles. When an event happens, i.e. when a transition *occurs*, tokens are removed from the *input places* for the transition, and new tokens are added to the *output places* for the transition. A place is an input place for a transition if there is an *arc* from the place to the transition. Output places are defined analogously. The *arc inscriptions* on the arcs connected to a transition determine which tokens are removed and added when the transition occurs.

In timed CP-nets, each token is allowed to carry a time value, also called a *time stamp*, in addition to the token value. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e., removed by the

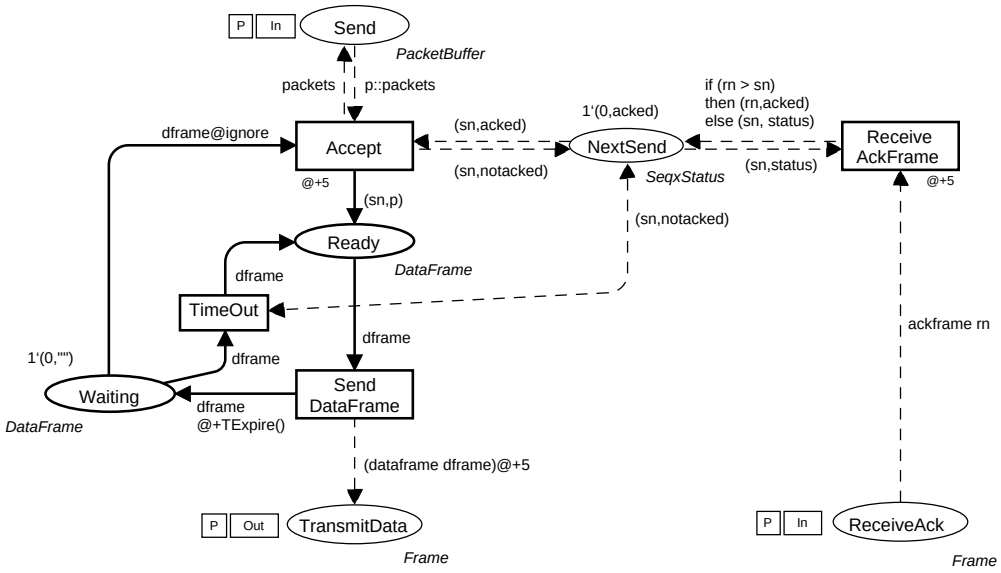


Figure 1.2: Sender page of stop-and-wait model.

occurrence of a transition. To model that an event takes r time units, we let the corresponding transition create time stamps for its output tokens that are r time units larger than the clock value at which the transition occurs. This implies that the tokens produced are unavailable for r time units. The execution of a timed CP-net is time driven, and it works in a way similar to that of event queues found in many languages for discrete-event simulation. The system remains at a given model time as long as there are transitions that can occur. When no more transitions can occur, the system advances the clock to the next model time at which transitions can occur.

1.3 Performance Analysis using Coloured Petri Nets

There is a large body of research concerning performance analysis using a variety of classes of Petri nets and Petri net-related formalisms. Most of this research focuses on solving analytical models that are automatically generated from the Petri net models. The size, complexity, and time concept for CP-nets prohibit the generation and solution of analytical models from CPN models. Therefore, performance analysis using CP-nets must rely on simulation to calculate performance measures for a model.

During a simulation of a CP-net, the CP-net can contain and generate quite a bit of quantitative information about the performance of a system, such as queue length, response time, throughput, etc. The previous section described how to model the states and events of a system using places, tokens and transitions. In other words, it described how to model the functionality of a system. Let us consider how CP-nets can be used to model and analyse the performance of a system.

Example When analysing the performance of a system, one is often interested in measuring the performance of system when it processes a particular kind of workload. For example, the workload for the stop-and-wait protocol is data packets, and when studying a bank the workload would be customers. With CP-nets it is possible to use both fixed workloads, i.e. workloads that are predetermined at the start of a simulation, and dynamic workloads. In the Upper Layers Send page, packets are generated on-the-fly during a simulation.

A timed CP-net can model how much time certain activities take. In most cases it is insufficient to model the average amount of time that a certain activity takes – it is necessary to include a more precise representation of the timing of the system. A user-defined function in the stop-and-wait model calculates an appropriate interarrival period each time a new packet arrives. The function is called each time a particular transition occurs. One model parameter determines whether the periods between packet arrivals are constant or exponentially distributed, and another parameter determines the average amount of time that passes between the arrival of two successive packets. Similar parameters and functions are used for determining the network delay and network reliability in the Communication Channel page.

For the stop-and-wait protocol, there are several performance measures of interest, including average queue length for the queue of packets waiting to be sent, average packet delay and network utilization. Packet delay is the time from which a packet is put in the Send buffer on the sender side until it is properly received by the receiver. Network utilization is the percentage of time in which the network is busy transmitting data frames or acknowledgement frames. The data channel is bi-directional, and the utilization of each direction of the channel can be measured. Changing the values of parameters mentioned above can have profound effects on the performance of the system.

Performance measures are calculated by observing and extracting data from the states and events of a CP-net during a simulation. Let us consider how the average queue length can be calculated for the stop-and-wait model. The queue of packets is modelled by a list of packets which is found on the place Send in the Sender page. The length of the queue changes either when a new packet is added to the list or when a packet is removed from the list. These events are modelled by the occurrence of the Generate transition in the Upper Layers Send page and the Accept transition in the Sender page, respectively. Therefore, the queue length can be measured accurately if the length of the list on the place Send is measured each time one of these two transitions occurs. The process of extracting the length of the queue from the state of the CP-net is referred to as *collecting data*. The length of the queue varies over time, therefore, the average queue length should be calculated as the *time-average* queue length. The time-average queue length is a weighted average of the possible queue lengths (0, 1, 2, ...) weighted by the proportion of time during the simulation that the queue was at that length. To calculate the time-average queue length during a simulation, the length of the list is weighted with the amount of time that passes before the length of the list changes. The sum of these values is calculated during a simulation, and the average queue length is equal to this sum divided by the length of the simulation (measured in modelled time units).

The necessary information must obviously be contained in the model in order to observe it. However, the user also needs a way to extract the data during simulation. The Design/CPN Performance Tool [87], which will also be referred to as the Performance Tool, can be used to extract and save data from CP-nets during simulations. Each performance measure is calculated by a user-defined *data collector*. A data collector consists of two user-defined functions: the *predicate function* determines when data is to be collected for a particular performance measure, and the *observation function* determines what data is to be collected. An observation function is invoked each time its associated predicate function returns true. Both functions can observe and extract data from all states that are reached and all events that occur during a simulation of a CP-net. Both functions must be written by the user, but template code can be generated to assist the user in creating the functions.

Figure 1.3 shows the two performance functions for a data collector named `Queue_Length` that calculates the time-average queue length for the stop-and-wait model. The predicate function, `Queue_LengthPred`, is invoked after every step in a simulation. It returns `true` whenever either the Accept transition or the Generate transition occurs. Otherwise, the predicate function returns `false`. When the predicate function returns `true`, the observation function, `Queue_LengthObs`, will be evaluated. Most of the code in Fig. 1.3 was generated automatically by the Performance Tool. The user only had to add `true` and `false` in lines 3-5 and all of line 14, which measures the length of the one list on place Send.

```
fun Queue_LengthPred (net_marking, binding_element) =                    1
  let                                                                    2
    fun filterFun (Bind.Sender'Accept (1, {sn,sent,packets,p,dframe})) = true   3
      | filterFun (Bind.UpperLayersSend'Generate (1, {packets,i})) = true      4
      | filterFun _ = false                                              5
  in                                                                     6
     filterFun binding_element                                           7
  end;                                                                   8
                                                                         9
fun Queue_LengthObs (net_marking, binding_element) =                     10
  let                                                                    11
    val mark_Send = PerfMark.Sender'Send 1 net_marking                   12
  in                                                                     13
      List.length (ms_to_col (striptime mark_Send))                      14
  end;                                                                   15
```

Figure 1.3: Performance functions for measuring queue length of packets.

The Performance Tool produces two kinds of simulation output. During a simulation, the data values that are extracted by observation functions can be used to calculate statistics and saved in *observation log files*. Figure 1.4 shows a *performance report* that contains statistics for five performance measures for the stop-and-wait model. The average queue length during that particular simulation was 18.15. Figure 1.5 shows how the length of the queue evolved

```
TIMED STATISTICS:
Name                 Count         Sum     Average     Maximum
------------------------------------------------------------
Queue Length          1150      907424       18.15          37
------------------------------------------------------------

UNTIMED STATISTICS:
Name                 Count         Sum     Average     Maximum
------------------------------------------------------------
Utilization            914        9184       10.05          18
Packet Delay           562       22230       39.56         234
Transmit Success      1627        1274        0.78           1
Time Outs              352         352        1.00           1
------------------------------------------------------------

Current step: 6030
Current time: 50006
```

Figure 1.4: Performance report.

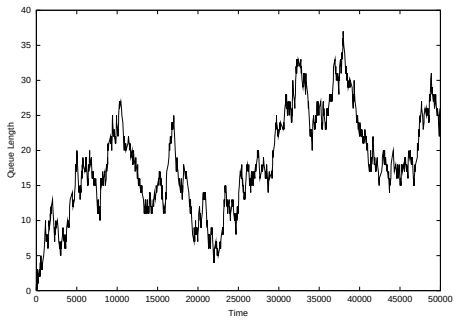


Figure 1.5: Graph of contents of an observation log file.

during the simulation that lasted for 50000 units of model time. The graph was obtained by plotting the contents of the observation log file that was maintained for the `Queue_Length` data collector.

1.4 Motivation and Aims of Dissertation

Coloured Petri nets is a general modelling language that have the potential for being used for performance analysis, but in practice it rarely is. As mentioned previously, other forms of Petri nets are frequently used for performance analysis, but they generally rely on analytical models which require that certain assumptions are made about the behaviour of the system. When using simulation-based performance analysis, fewer assumptions need to be made about the system in order to model or analyse its behaviour.

We have seen an example of how a CPN model can contain and generate quite a bit of quantitative information about the performance of the model during a simulation. Until recently, the information was neither easily accessible nor frequently used. To help remedy this problem, Bo Lindstrøm and I designed and implemented the Design/CPN Performance Tool which was the main topic of our Master's thesis [88], and which we implemented while we were working as part-time student programmers. The Performance Tool was put to practical use in several projects, and it was fully integrated into version 4.0 of Design/CPN which was released in September, 1999.

The goal of the work in this dissertation has been to further investigate and facilitate the practical use of CP-nets for performance analysis of realistic, industrial-sized systems. Coloured Petri nets are well suited for modelling and analysing real-world systems. This, in fact, is one of the goals of the CPN Group at the University of Aarhus:

> "The development of CPN has been driven by the desire to develop an industrial-strength modelling language – at the same time theoretically well-founded and versatile enough to be used in practice for systems of the size and complexity found in typical industrial projects." [80]

To achieve this goal the developers maintain that it is important to support research in all aspects of CP-nets, i.e. theoretical foundation, tool support, and large-scale practical applications. The work presented in this dissertation continues in this spirit.

One aim of this dissertation is to introduce simulation-based performance analysis to the world of CP-nets. In the past, simulation has most often been used for debugging, validation, and for investigating logical correctness of systems. There are relatively few studies concerning performance analysis using high-level Petri nets, and there seems to be only very limited research in the area of using high-level Petri nets for simulation-based performance analysis. This is somewhat surprising since there is an incredibly active simulation community, as evidenced by the number of conferences, journals and societies related to simulation (see, for example, [66] for a collection of simulation-related links). However, there seems to be very little overlap between the simulation community and the Petri net community

In order to use CP-nets for performance analysis, it must be possible to collect data from CP-nets during simulation. This, in turn, means that the necessary data must be accessible in the model, and there must be support for extracting data and generating reliable simulation output. The Design/CPN Performance Tool provides reasonable support for extracting data during simulations, but as we shall see, it provides only very limited support for proper simulation output analysis. Cleaner, more understandable CPN models can be created if there is an explicit separation between modelling the behaviour of a system and observing the behaviour of the model. In other words, it should not be necessary to add places or transitions or to modify net inscriptions for the sole purpose of extracting information regarding the performance of the model.

1.5 Outline of Dissertation

This dissertation consists of two parts. The main body of work done during the course of my PhD studies is documented in four papers [90, 89, 127, 126]. All four of these papers have been accepted for presentation at international workshops. Part II of this dissertation (Chapters 7-10) consists of these four papers. The publication history and status for each of the papers can be found below and at the start of the appropriate chapter.

Part I (Chapters 1-6) of this dissertation constitutes the obligatory overview paper which summarises the work described in the papers in Part II of the dissertation. The remainder of Part I is organised as follows:

Chapter 2 summarises the paper *Performance Analysis using Coloured Petri Nets* [126]. The paper will appear in the proceedings of the Tenth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02). The paper that was submitted to MASCOTS'02 is contained in full in Chapter 7, a shorter version of the paper will appear in the proceedings of MASCOTS'02. The paper contributes to the area of tool support for CP-nets.

Chapter 3 summarises the paper *Towards a Monitoring Framework for Discrete-Event System Simulations* [90]. The paper presents joint work with Bo Lindstrøm, and it will appear in the proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02). The paper is contained in full in Chapter 8. The paper contributes to the area of tool support for CP-nets.

Chapter 4 summarises the paper *Annotating Coloured Petri Nets* [89]. The paper presents joint work with Bo Lindstrøm, and it will appear in the proceedings of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'02). The paper is contained in full in Chapter 9. The paper contributes to the area of CP-net theory.

Chapter 5 summarises the paper *Simulation Based Performance Analysis of Web Servers* [127]. The paper presents joint work with Søren Christensen and Lars M. Kristensen and Kjeld Mortensen, among others. The paper has been published in *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 59-68, IEEE, 2001. The paper is contained in full in Chapter 10. The paper contributes to the area of practical applications of CP-nets.

Chapter 6 concludes the overview paper by summarising the main contributions of the dissertation and by discussing directions for future work.

Chapters 2-5 each consist of three sections. The first section provides, if necessary, an introduction to the specific topic(s) addressed in the paper in question. In some cases, the first section will also provide background information regarding the context in which the work was done. The second section provides a summary of the paper in question, together with a brief evaluation of the methods and techniques that were used. The third section provides a survey of related work and compares the results in the paper with the related work.

Readers' Guide Detailed knowledge about CP-nets is not required to understand this overview paper, which constitutes Part I of this dissertation. The reader is assumed to have a basic understanding of Petri nets. It is, however, an advantage to have a good understanding of CP-nets when reading Part II of this dissertation. Readers without knowledge of CP-nets and who wish to study this dissertation in more detail are referred to "The Practitioner's Guide to Coloured Petri Nets" [80] which provides a practical introduction to coloured Petri nets. Furthermore, readers who are interested in understanding the formal definitions and proof in Chapter 9 would benefit from a firm understanding of CP-nets as defined by Jensen in [70]. No statistical knowledge is required. Any statistical terminology that is introduced will be informally described, and Sect. 2.1.2 provides a brief introduction to some of basic concepts related to simulation-based performance analysis. Readers who are interested in learning more about output-data analysis and experimental design methods that are

applicable to real-world problems are referred to Law & Kelton's well-known *Simulation Modeling & Analysis* [84].

The papers in Chapters 7-10 can be read in any order. They have, however, been organised in the order in which, I feel, they make the most significant contributions towards facilitating the practical use of CP-nets for performance analysis of industrial-sized systems. Tool support for conducting reliable, simulation-based performance analysis using CP-nets is presented in [126]. Facilities which support explicit separation between modelling the behaviour of a system and observing the behaviour of a model are presented in [90]. These facilities support the creation of cleaner and more understandable CP-nets. A method for introducing auxiliary information into CP-nets without affecting the behaviour of the CP-net is the topic of [89]. The addition of such information can often aid in analysing the performance of a CP-net. Finally, the case study presented in [127] discusses how CP-nets and the newly developed performance facilities can be used to analyse the performance of a web server.

This dissertation addresses three, very large research areas: (coloured) Petri nets, performance analysis, and simulation analysis. In Part I of this dissertation, the term *simulation analysis* covers both output-data analysis and experimental design, i.e. the determination of which simulation experiments should be executed. My expertise is primarily within the area of CP-nets. During my PhD studies, I have obtained a basic understanding of the other two areas, but I am, by no means, an expert in either of these two areas. When discussing work that is related to my own, it would be impossible to report on the state-of-the-art within all three fields. I will approach the discussion of related work from a CPN point-of-view. However, I will also discuss my work in the broader contexts of simulation analysis and performance analysis.

Chapter 2

Performance Tools for Coloured Petri Nets

This chapter discusses the paper "Performance Analysis using Coloured Petri Nets." Section 2.1 discusses the motivation behind the further development of performance facilities for Design/CPN, and it provides an informal introduction to concepts related to simulation-based performance analysis. Section 2.2 summarises the main contributions of the paper and evaluates the techniques that were applied. Section 2.3 discusses related work.

2.1 Introduction and Background

The goal of the work on which this paper is based was to improve support for simulation-based performance analysis using CP-nets. With the Performance Tool (described in Sect. 1.3) it is fairly easy to extract data, calculate some statistics, and save all data observations during a simulation of a CP-net. The facilities were intentionally designed to be general and flexible. However, for standard data collection and analysis purposes, they require too much effort on the part of the user: all data collectors must be defined by a user, there is no standard support for automatically running multiple simulations, and care must be taken to ensure that simulation output is not deleted when a new simulation is run. The tool provides high-level support for data collection from CP-nets, but it lacks support for reliable statistical analysis of the resulting simulation output.

2.1.1 Motivation

Simulation analysis is not simply an exercise in computer programming and data collection. According to simulation experts [69, 84] some of the common drawbacks to simulation studies are:

- Statistical techniques are used improperly
- There is a tendency to have greater confidence in the results of a study if a large volume of numbers are produced during the study
- A great deal of time is often spent developing the model, but little effort is made to properly analyse the output

- Analysts are experts in the use of modelling tools or in the domain of the system being modelled, but they do not know how to analyse or interpret data

These were precisely some of the problems that were encountered when using the Performance Tool in various projects. With the Performance Tool, it was suddenly possible to generate huge amounts of data, but the users of the tool were inexperienced data analysts, and the data that was produced during a single simulation was often interpreted to be *the* answer.

A message that appears repeatedly in performance analysis literature is that performance analysts should know something about probability, statistics, and statistical analysis techniques. In my experience, both at the University of Aarhus and as a visitor at the University of California at Los Angeles, it is easy to disregard or to be unaware of this seemingly obvious recommendation. Unfortunately, quite sophisticated and elegant performance modelling tools can be used for questionable simulation studies. At first glance, some simulation studies appear quite sound, but upon closer inspection the studies are unconvincing. Often there is no apparent systematic approach to defining experiments, and values for system parameters seem to be chosen arbitrarily. Furthermore, it is often unclear whether proper statistical techniques are used to analyse output data. These problems can be avoided, to some extent, by supporting sound statistical techniques directly in modelling tools.

I had the rather unique opportunity to take, and then a year later, team-teach a course on simulation, modelling and analysis with a particular emphasis on data analysis techniques. When I took the course, it was only offered for students in the Operations Research (OR) Department. Through this course I learned about the basics of what data analysts need for doing proper statistical analysis of simulation output. When I was involved in teaching the course, it was offered jointly by the OR and Computer Science (CS) departments. In this course, the students primarily used the Arena simulation package [75, 5] for modelling and analysing a variety of different systems. The course included a brief introduction of CP-nets to expose the students to an alternative modelling language.

My involvement in these two courses has been a valuable learning experience. I observed that the students from different backgrounds approached their modelling and analysis projects very differently. The OR students were very concerned with designing good simulation studies and doing proper analysis of the simulation output, and they were generally not as critical when it came to validating the functionality of their models. The CS students, on the other hand, were very concerned with creating valid models, but their analysis of their simulation output was generally weak. These experiences have provided valuable insights into the needs of different users. Inexperienced data analysts will have a tendency to believe what a tool tells them, therefore care must be taken to avoid generating misleading simulation output. More experienced data analysts generally require that more sophisticated kinds of data are generated for specific purposes.

The typical user of CP-nets is not likely to be an expert data analyst, and this has influenced the further development of performance facilities for CP-nets. By providing simple output-analysis facilities, a user is allowed to focus on the analysis of the data rather than the manipulating or post-processing of the data. The kind of simulation output that is generated may also provide some hints regarding the proper analysis of the data. New performance facilities should assist a CPN user in a simulation study rather than lead an inexperienced data analyst directly into the major pitfalls associated with simulation studies. New performance features have been designed such that users have an opportunity to use the facilities without being an experienced programmer or an expert CPN user. On the other hand, the features are still general enough that experts will be able to tailor the performance facilities to their needs.

2.1.2 Basic Concepts Related to Simulation

Before presenting the new performance facilities, this section will provide an informal introduction to basic concepts related to simulation-based performance analysis. References will be provided for more formal and precise definitions of each of these concepts. Simulation output data are random, therefore appropriate statistical techniques must be used both to design and and to interpret simulation experiments.

Terminating and Non-terminating Systems Simple statistical techniques exist for using simulation models to analyse *terminating* and *non-terminating* systems[1] which are defined in turn below.

Terminating systems are characterised by having a fixed starting condition and a naturally occurring event that marks the end of the system. An example of a terminating system is a work shift that starts at 8 am and ends at 4 pm at a car assembly plant. For terminating systems the initial conditions of the system generally affect the desired measures of performance. The purpose of simulating terminating systems is to understand their behaviour during a certain period of time, and this is also referred to as studying the *transient behaviour*[2] of the system. *Terminating simulations*[3] are used to simulate terminating systems. The length of a terminating simulation is determined either by the system itself, if the system is a terminating system, or by the objective of a simulation study. For example, the goal of a performance study may be to analyse the performance of the work shift at the assembly plant either during the whole shift or only during the time it takes to assemble the first ten cars after the work shift has started. The length of a terminating simulation can be determined by a fixed amount of time, e.g. 8 hours, or it can be determined by some condition, e.g. the completion of the tenth car.

In a non-terminating system, the duration of the system is not finite. The Internet exemplifies a non-terminating system. *Non-terminating simulations* are used to simulate non-terminating systems. In a non-terminating simulation,

[1] For more details regarding (non-) terminating systems, see, e.g., pg. 27 in [13].
[2] For more details regarding transient and steady-state behaviour, see, e.g., Sect. 9.2 in [84].
[3] For more details regarding (non-) terminating simulations, see, e.g., Sect. 9.3 in [84].

there is no event to signal the end of a simulation, and such simulations are typically used to investigate the long-term behaviour of a system. Non-terminating simulations must, of course, stop at some point, and it is a non-trivial problem to determine the proper duration of a non-terminating simulation. If the behaviour of the system becomes fairly stable at some point, then there are simple techniques for analysing the *steady-state behaviour* of the system using non-terminating simulations. When analysing steady-state behaviour using non-terminating simulations, it is often useful to be able to specify a *warmup period*[4] in which data is not collected because the model has not yet reached a steady state. Determining when, or if, a model reaches steady state is also a complicated issue.

Estimating Performance Measures Performances measures that are calculated via simulation are generally only *estimates* of the true performance measures. *Confidence intervals*[5] can be used to indicate how precise estimates of a performance measure are. In order to calculate accurate confidence intervals, it must be possible to collect estimates that are independent and identically distributed[6] (IID). Intuitively, performance measure estimates are IID if they are not related to each other and if they have the same probability distribution. Let us consider examples of estimates that are not IID. The amount of time that customer number n waits in a queue at a bank is not likely to be independent from the amount of time that customer number $n-1$ waits in the queue, because customer n must often wait behind customer $n-1$, i.e. the queue delays for these two customers are *not* independent. The average arrival rate for customers at a bank from 9-10 am and the average arrival rate for customers from 12-1 pm are probably not identically distributed since more customers are likely to come during lunch hour. When studying terminating systems, IID estimates of performance measures must be collected from a number of independent, terminating simulations. Let us consider examples of IID estimates. If one estimate of the average queue delay of bank customers is calculated for each of a number of terminating simulations, then these estimates of the average queue delay *are* independent. Similarly, if the average arrival rate of customers between 9-10 am is calculated for each terminating simulation, then these estimates are likely to be identically distributed. When studying steady-state behaviour, there are several different techniques for collecting IID estimates from both terminating and non-terminating simulations.

Variance Reduction Techniques One of the drawbacks of simulation-based performance analysis is that it can take a long time to run one simulation. This problem is amplified if many simulations need to be run. *Variance-reduction techniques* [7] (VRT) can sometimes be used to reduce the number or length of simulations that need to be run. The variance reduction techniques known as

[4]For more details regarding warmup, see, e.g., Sect. 9.5.1 in [84].
[5]For more details regarding confidence intervals, see, e.g., Sect. 4.5 in [84].
[6]For more details regarding IID data, see, e.g., p.12 in [84].
[7]For more details regarding variance-reduction techniques, see, e.g., Chapter 11 in [84].

common random numbers (CRN) and *synchronisation* are particularly useful when comparing alternative system configurations. With CRN, the same random numbers are used in each configuration, and synchronisation is achieved if each random number is used for the same purpose in each configuration. Not only can CRN reduce the number or length of simulations that need to be run, but it also implies that a fairer comparison of the configurations can be made because the experimental conditions are the same for each configuration [54].

2.2 Main Contributions

Th paper "Performance Analysis using Coloured Petri Nets" presents new facilities for performance analysis using CP-nets. The paper focuses on how CP-nets can be used to analyse network protocols, but the facilities that are discussed can be used to analyse any kind of system. Network protocols are particularly interesting because it is often important to analyse both the functionality and the performance of a protocol, and CP-nets is a formalism that can be used to analyse these two aspects of a system. The model from Sect. 1.2 is used as an example in the paper. The facilities have been designed to assist inexperienced data analysts.

The first contribution of the paper is to present practical applications of so-called *monitors* when using CP-nets to study the performance of systems. A monitor is a mechanism that can observe (or monitor) the states and events of a CP-net during a simulation, and that can take appropriate actions based on the observations. Monitors can be used both to inspect and to control a simulation. Monitors are a concrete realisation of the *monitoring framework* that is discussed in Chapters 3 and 8.

The paper discusses several kinds of monitors that have been implemented. A *simulation breakpoint monitor* controls a simulation of the stop-and-wait model by stopping a simulation when a packet is retransmitted for the third time. This example also illustrates how several different monitors can interact in order to achieve a common goal. Currently, Design/CPN and CPN Tools only provide support for properly stopping a simulation when either a given number of steps have occurred or when a given amount of model time has passed. While it is possible to use workarounds, e.g. by raising exceptions, to stop a simulation, it is not recommended since valuable data can be lost when a simulation is unexpectedly interrupted. Simulation breakpoint monitors should prove to be extremely useful for defining domain-specific or model-specific simulation breakpoints. A monitor for creating message sequence charts [67] (MSC) illustrates how the performance of a network protocol can be visualised.

Data collection monitors represent the data collection facilities that are being developed for the new simulator for Design/CPN and CPN Tools. These monitors are similar to the data collectors in the Performance Tool. As with all other monitors, data collection monitors can extract data from the states that are reached and the events that occur during a simulation of a CP-net. New *standard monitors* can be used, e.g., to calculate the average number of tokens on a place or to count the number of times a particular transition occurs during

a simulation. Standard monitors are largely predefined and model-independent. Support, in the form of template code, is also provided for defining more specialised or model-dependent data collection monitors.

The second contribution of the paper is the introduction of facilities supporting statistically reliable analysis of one configuration of a CP-net. Two of the major pitfalls in simulation studies are 1) regarding the results of a single terminating simulation as the "true answers" and 2) the improper use of statistical techniques. Several new features have been added in an attempt to avoid these problems. A new *batch script*, which is just an ML function, will run a given number of independent, terminating simulations, and data is automatically collected and saved during each simulation. New *batch data collection monitors* can be used to calculate confidence intervals for IID performance measure estimates that are collected from independent simulations.

Simulation output is crucial for performance analysis. It is used both for analysing the performance of the system and for presenting the results of the analysis. Therefore, it is important that a simulation modelling tool generates output that is useful for data analysts. The individual data values that are observed by a data collection monitor can be saved in observation log files. Simulation performance reports and new *batch performance reports* contain statistics that are calculated for single simulations and batches of simulations, respectively. Both types of performance reports can be now saved in two new formats (LaTeX and HTML) in addition to plain text. Additional facilities can be used to save all simulation output in a simple, yet organised directory system. A *batch status file* provides information about the status of each individual simulation, including number of steps taken, model time at the end of the simulation, reason why the simulation stopped, and the directory in which the output was saved. Two kinds of scripts for plotting the contents of observation log files with gnuplot [53] can now be generated. The IID performance measure estimates from the independent simulations are also saved in a file, which can be post-processed by the user.

The third contribution of the paper is the introduction of facilities that support the analysis of several different configurations of a CP-net. Simulation studies can be made for many different reasons, including analysing the performance of one system, comparing the performance of several given configurations, locating the model parameters that have the most significant impact on a particular performance measure, or finding the combination of parameters that gives the best results. Inherent in most of these activities is the need to be able to run multiple simulations for different system configurations. Another new batch script will automatically run a given number of simulations for different system configurations, assuming that the different configurations can be specified by changing numerical parameters in a CP-net.

Support has also been added for using the variance reduction technique of common random numbers and synchronisation. The paper provides an example that illustrates the effects of using varying degrees of CRN and synchronisation when comparing alternative system configurations. It is shown that comparing two configurations is relatively easy using standard simulation output to

calculate *paired-t confidence intervals*[8]. Paired-t confidence intervals indicate whether the performance of two configurations are significantly different.

The new facilities that are presented in this paper should prove to be useful when using CP-nets for performance analysis. The typical CPN user is likely to be neither an experienced performance analyst nor an experienced data analyst. The facilities have been designed with this in mind, and they have been designed to aid and assist inexperienced data analysts. All of the statistical techniques that are discussed in this paper are standard, well-known, and relatively simple techniques, and there is plenty of room for improvement. For example, there must be better support for running and analysing non-terminating simulations, but this can be achieved by making some straightforward changes to the batch script for running a number of independent simulations. These facilities have been implemented and have undergone preliminary testing. However, it is clearly important that the various facilities should be improved and integrated into one CPN tool.

2.3 Related Work

The topic of the paper is facilities for modelling and analysing the performance of systems. The area of related work is virtually limitless. Therefore, in this section I will limit the discussion of related work primarily to a survey of the prominent results within the area of (simulation-based) performance analysis using Petri nets. A brief comparison will be made of performance tools for Petri nets and a widely-used commercial simulation package (Arena) in order to provide a sense of the similarities and differences between these two kinds of tools.

2.3.1 Petri Nets for Performance Analysis

There are many different classes of Petri nets that are used for performance analysis, and they all share the common trait of being timed models. Most of the current research concerning Petri nets and performance analysis focuses on using low-level Petri nets, such as Stochastic Petri nets [93, 10] (SPN), Generalized Stochastic Petri Nets [1] (GSPN), Deterministic Stochastic Petri Nets [86] (DSPN), and the closely related Stochastic Activity Nets [112] (SANs). Each of these kinds of Petri nets require that some, if not all, transitions have exponentially distributed firing delays and atomic firing policy. Under certain circumstances, alternative firing delays can be used in, e.g. GSPNs and DSPNs. The most well-known and widespread PN and PN-related tools for performance analysis are GreatSPN [29, 57], DSPNexpress [85, 42], TimeNET [51, 119] and UltraSAN [120, 113]. In the following, I will refer to these tools as the *PN performance tools*. Stochastic well-formed nets [28] (SWN) are are a high-level variant of GSPNs in which there are tight restrictions on which data types may be used to define token values and on the kinds of arc inscriptions that are

[8]For more details regarding paired-t confidence intervals, see, e.g., Sect. 10.2 in [84].

allowed. SWNs are also supported in the GreatSPN tool. These are the most commonly used tools and Petri net classes in the area of performance analysis and Petri nets.

Analytical models can be automatically generated from the Petri nets mentioned above. The analytical models can then be solved using sophisticated techniques to give exact values for the performance measures that have been defined for the net in question. Many analytical models require that it is possible to generate a full state space[9] for the system in question. One problem that can arise when using state spaces to derive performance measures is connected to the *state explosion problem* [121] which means that even for small configurations of a system the number of reachable states may be very large or even infinite. When the state space of a system is too large, then it may be difficult, if not impossible, to generate the analytical models needed for performance analysis. In some cases the state explosion problem can be avoided by using small and unrealistic configurations of the system, but this is not desirable if the goal of a study is to analyse a realistic configuration of an industrial-sized model. If an analytical model can be generated and solved, then it is certainly advantageous to be able to calculate exact performance measures, but the reliability of the results may be compromised if unrealistic assumptions, such as exponentially distributed time delays, need to be made about the system in order to model it with Petri nets.

Simulation is always an alternative when analytical models cannot be derived for the types of Petri nets that have been mentioned here. The PN performance tools have long histories in supporting both analytic and simulation-based performance analysis using Petri nets. In addition to Design/CPN and CPN Tools, the Ex*Spect* [122, 43] and Artifex [7] tools also support simulation of CP-nets. As we have seen, it is possible to use CP-nets and simulation for performance analysis. An advantage of using CP-nets for performance analysis is that fewer assumptions need to be made regarding the behaviour of the system being modelled, e.g. time delays do not have to be exponentially distributed. Furthermore, the state explosion problem is not an issue when using simulation. The main drawback of simulation is that it can only be used to estimate the performance of a model. Additional drawbacks of simulation-based performance analysis are discussed below.

2.3.2 Simulation Analysis

As mentioned previously, in this dissertation the phrase *simulation analysis* covers both output-data analysis and experimental design, i.e. the determination of which simulation experiments should be executed. This section discusses how the established PN performance tools support simulation analysis.

Data Analysis There are actually two types of data analysis associated with simulation studies: *input-data analysis* and *output-data analysis*. Input-data analysis typically involves fitting raw data from the domain of the study to

[9] A full state space is essentially a graph representing all of the reachable states and state changes of a system.

probability distributions which can then be used to represent the corresponding aspect in the model. Examples of software that can be used for input-data analysis can be found in [117]. This is an important topic and is relevant for many simulation studies, but it is beyond the scope of this dissertation.

Output-data analysis is concerned with the analysis of data that is generated during simulations. The reliability of a simulation study depends on the use of sound statistical analysis techniques. Most of the PN performance tools can run terminating and non-terminating simulations, and performance measures can be calculated for transient and steady-state behaviour. In the PN performance tools, the method for stopping all terminating simulations is to stop them after a certain amount of time has passed. In other words, the tools do not support the use of model-specific stop criteria. In many of the tools mentioned, it is possible to specify the length of the warmup period when running non-terminating simulations. In some cases, TimeNET can even automatically detect the proper length of the warmup period. When the user specifies the warmup period, there is no guarantee that the period specified actually corresponds to a reasonable warmup period. If the individual data observations from a number of independent simulations are available, a graphical procedure [84], due to Welch, can be used to estimate the proper length of the warmup period. The new performance facilities for CP-nets provide basic support for running and analysing terminating simulations, but the most obvious weakness of the facilities is the lack of explicit support for analysing steady-state behaviour.

Most simulation software has capabilities for providing 95% confidence intervals, and this holds true for the PN performance tools. Many of the tools can also calculate 90% and 99% confidence intervals, and the confidence intervals can be calculated in order to obtain a user-specified absolute or relative precision. The confidence intervals that are presented in my paper are calculated using a fixed-sample size, i.e. the number of simulations that were run. This is easy to implement, but it is not very flexible since an analyst cannot specify how precise the confidence intervals should be.

Simulation Output Simulation output it crucial for simulation-based performance analysis. It is used both for analysing the performance of the system and for presenting the results of the analysis. Therefore, it is important that a simulation modelling tool generates output that is useful for data analysts.

The PN performance tools save the calculated performance measures in files. Some of the tools can also dynamically update the values of performance measures in the GUI of the tool during a simulation. Many of the tools also use gnuplot to create and save various kinds of graphs of performance measures. In some cases the user is able to specify some aspects of the graphs, such as labels and scales, while other tools use predetermined formats. None of these tools appear to save the individual data values that are used to calculate performance measures during a single simulation. However, TimeNET can dynamically graph the evolution of performance measures during one simulation. While the data values from one terminating simulation can generally not be used to reliably estimate a performance measure, they can be extremely useful

for visualising the behaviour of the system during the simulation. For example, plotting the evolution of the length of a queue during a simulation can be quite helpful , e.g. when discussing the behaviour of the model with non-PN experts, when debugging the model, and when estimating the warmup period.

Comparing Configurations Simulation is often used to compare alternative system configurations, therefore it is useful if there is support for defining and automatically simulating alternative configurations. In the PN performance tools that provide support for comparing different configurations, different configurations are defined by changing the values of numerical parameters. Some tools can only automatically vary the value of one parameter, while other tools are able to automatically vary a set of numerical parameters.

The new performance facilities for CP-nets also contain support for defining and simulating configurations that are specified by changing numerical parameters in a CP-net. A advantage of CP-nets is that there is the potential for varying the values of non-numerical parameters, when defining different configurations. Examples of non-numerical parameters are queueing or caching strategies that are declared as colour sets, or names of input files that contain predefined workload. However, the PN performance tools have the added flexibility that both simulation and analytical models can be used to analyse each of the configurations.

After defining and simulating different configurations, it is necessary to compare the performance of the different configurations. As usual, only sound statistical techniques should be used to evaluate whether there is a significant difference between the performance of two or more systems. When comparing configurations with GreatSPN, one estimate of each performance measures for each configuration are saved in one file. Several PN performance tools generate graphs that can provide a sense of whether two configurations are significantly different. However, since neither the results from GreatSPN nor the graphs contain an indication of how precise the estimates of performance measure are, there is no guarantee that the performance of the configurations are actually significantly different. The graphing facilities in the PN performance tools require little effort on the part of a user, but in some cases the comparisons of two configurations will be inconclusive based on the information that is available. The new performance facilities for CP-nets require more work on the part of the user because the user must always post-process the data, e.g. to calculate paired-t confidence intervals. However, by post-processing data it is possible to conclude whether or not there is a significant difference based on the available data.

Rare Events Some performance studies deal with understanding the behaviour of systems in extreme situations. For example, the goal of the study may be to examine what happens when a vital component in the system fails, or when the system experiences a sudden unexpected burst in incoming workload. Under normal circumstances, these events occur only rarely, and this can complicate simulation studies because extremely long simulations need to

be run in order to collect sufficient amounts of data regarding the rare events. One case study [94] used CP-nets to study failures and so-called feared events, which are both rare events, in oil tanks in cars. In this study, long simulation times were avoided because it was possible to translate a simple CP-net of the oil tank to a recursive ML function with which the performance of the oil tank could be observed. It was up to 500 times faster to evaluate the ML function than to run a simulation of the corresponding CP-net. Presumably it is very rare that a simple ML function can be derived from a CP-net, in which case better support is needed for studying rare events in CPN simulations.

Special techniques are supported in UltraSAN [98] and TimeNET [74] for alleviating the problem of long simulation times when studying rare events. Previously intractable simulations may become tractable when employing these techniques, thus allowing the analysis of new kinds of systems.

2.3.3 Commercial Simulation Package

PN performance tools may provide state-of-the-art methods for generating and calculating analytical models, but they do not provide state-of-the-art support for simulation-based performance analysis. This section briefly compares PN performance tools with a commercially available simulation package. Arena has been chosen as a representative example of the commercially available, graphically-based simulation tools. Personal experience with the package is one obvious reason for choosing Arena. However, since Arena is licensed to over 5000 customers and is used by over 500 educational institutions [5], it seems fair to assume that it is one of the widely-used simulation packages.

Arena provides support for all aspects of simulation modelling and analysis. Models are generally built using drag-and-drop graphical representations of predefined modules, such as queues, servers, transport entities, and logical modules (e.g. if-statements and while-loops). If input data is available for the system being modelled, then Arena can fit the data to a probability distribution which can then be used to represent the data in the model. Both terminating and non-terminating simulations can be run for studying transient and steady-state behaviour. Output reports contain statistics for standard and user-defined performance measures. All data that is collected during simulations is saved in a database. The database is accessed by a data-analysis component in Arena that can, e.g. post-process data, plot graphs of data from single simulations, and compare different configurations using reliable statistical techniques. A user can manually define a set of different model configurations that should be simulated and analysed. Arena can also automatically search for an optimal configuration of a system.

While the performance facilities in Arena are more advanced than the performance facilities in any of the PN performance tools, the package also has its drawbacks. The modelling language in Arena is a general language that is not theoretically well-founded. Therefore, there is no support for verifying the logical correctness of an Arena model. (Coloured) Petri nets have the advantage that they can be used for both performance and functional analysis of systems. Another drawback to Arena is that some of the predefined modules are quite

complicated. When using the textbook introduction to Arena version 3.0 [75], a user is shown how to build rather complicated models quite fast. While this provides instant satisfaction, problems arise when inexperienced users attempt to build models of systems that are significantly different from the examples in the book. It may take slightly longer to learn how to build (C)PN models, but once a user knows the basics, it becomes possible to model many different kinds of systems.

Chapter 3

Monitoring Simulations

monitor (*verb*) to watch, keep track of, or check usually for a special purpose

- Merriam-Webster's Collegiate Dictionary

This chapter discusses the paper "Towards a Monitoring Framework for Discrete-Event System Simulations." Section 3.1 introduces the concept of monitoring simulations. Section 3.2 summarises the main contributions of the paper and evaluates the techniques that were applied. Section 3.3 discusses related work.

3.1 Introduction and Background

The basic tasks for simulators of CP-nets are: to calculate the initial marking, to locate the transitions that can occur in each marking, to select the transition that should occur next, and to transform the current marking to the next marking according to the occurrence rule. In order to analyse the performance of a CP-net, it is necessary to carry out additional tasks during a simulation. At a minimum, it must be possible to observe the markings that are reached during a simulation, to extract data from the markings, and to calculate statistics from the data. If the data is to be saved in a file, then the file needs to be opened, updated, and closed at appropriate points during the simulation.

More advanced support for performance analysis can be provided if additional functionality is supported. For example, additional performance measures can be calculated if occurring transitions can also be observed. Moreover, accurate stopping criteria for terminating simulations can be defined if a simulator supports additional functionality. CPN simulators can generally stop a simulation after a certain number of steps have been taken or after a certain amount of model time has passed. When running terminating simulations it is often useful to be able to define domain-specific simulation stop criteria. For example, a simulation study may be interested in measuring the average packet delay for the first 100 packets in the stop-and-wait model from Sect. 1.2. Supporting functionality for visualising the performance of a CP-net during a simulation, using e.g. MSCs or domain-specific graphics, also has several advantages: it is a quite useful debugging technique, it is helpful during initial

attempts to find reasonable parameter values, and it becomes possible for non-CPN experts to understand and discuss the behaviour of the CP-net.

We introduce the term *monitoring* to generically describe any activity related to observing, inspecting, controlling or modifying a simulation of a CP-net. In other words, a process *monitors* a simulation if the process examines a CP-net during the simulation, periodically extracts information from the CP-net, and uses the information for a specific purpose. Examples of processes that monitor CPN simulations are the mechanisms that collect data or update MSCs in Design/CPN.

A simulation of a CP-net can be monitored in two different ways. The first option is to take advantage of the expressiveness of the CPN formalism and the inscription languages and encode monitoring functionality directly in a CPN model. This can be achieved by adding extra places and transitions to the net and by using functions that have side effects. Encoding monitoring functionality directly in a CP-net has the advantage that it does not require any extra effort on the part of developers of CPN simulators. However, adding extra net structure or complicated functions may affect the behaviour of the model in unexpected or undesirable ways. In addition, incorporating extra functionality in a model is time consuming for a modeller.

An alternative that would avoid the problem of introducing unwanted behaviour into a CP-net is to extend the functionality of the simulator with explicit support for monitoring. This, of course, requires more work for the developers of a simulator. The main advantage is that it supports the creation of cleaner, more understandable models.

A number of libraries and tool extensions have been developed for monitoring simulations in Design/CPN during the past decade. These libraries include support for creating MSCs [92], updating domain-specific graphics [106], communicating between the simulator and other processes [49], and collecting data [87]. These libraries and tool extensions can be used both to inspect and to control a simulation of a CP-net. In Design/CPN, monitoring activities are often encoded in *code segments* which are associated with transitions. A code segment contains arbitrarily complex SML code that is evaluated when the corresponding transition occurs. In this way, it is possible to access monitoring facilities when a transition occurs. Many published CPN models also include net structure, inscriptions, or entire pages that do not model any aspect of the modelled system. These extra elements are used solely to execute auxiliary (non-CPN) functionality, such as opening files, generating complicated initial markings, or collecting data and calculating statistics. Until recently, code segments have been the only means for separating modelling the behaviour of a system and monitoring the behaviour of a model in Design/CPN.

3.2 Main Contributions

The main contribution of this paper is a general framework for designing and implementing monitoring facilities in CPN tools. We use the term *monitor* to denote any mechanism which inspects or monitors the states and events of a

```
if check (event, state)
    then act (observe (event, state), state)
```

Figure 3.1: General functionality of a monitor.

CPN model, and which can take an appropriate action based on the observations. A simpler version of the stop-and-wait model is used in the paper to illustrate how monitors can be created and used. One of the goals of the monitoring framework is to make it possible to use monitors to inspect or control a simulation without having to alter models.

Based on our experiences, we have identified general patterns in how ad hoc monitoring is done. In general, each monitor can be divided into three logical parts. The first part *checks* when the monitor should make an observation. The second part *observes* a specific value from the model. For the third part, each different kind of monitor *acts* in a particular manner based on the observation made. Figure 3.1 illustrates the proposed general functionality of a monitor. In some cases it may also be useful for a monitor to be able to initialise some values and conclude some work after monitoring a simulation.

During a simulation, a monitor should be able to inspect the current state of the model and the most recently occurred event during a simulation. In addition to the current state and the most recently occurred event, it is useful if the monitor has access to the simulator of the model, and to the simulator environment. Access to the simulator makes it possible to, e.g. stop a simulation, while access to the simulator environment may give additional possibilities such as accessing global variables, functions, and file systems. If monitors are able to act on the state of the model, then care must be taken because the semantics of CP-nets could be violated if used improperly. The paper also provides a proposal for the complete interface of monitors. The interface includes three functions named `check`, `observe`, and `act`.

There are several advantages of using this common framework for creating monitors within a given tool. The main advantage is that it becomes possible to make an explicit separation between modelling the behaviour of a system and monitoring the behaviour of the model. Support for monitors can be implemented directly in simulators, and the simulator can automatically activate monitors at appropriate points during a simulation. As a result, it should not be necessary to alter models in order to monitor them during simulations. Another advantage is that by having a common interaction technique for all monitors it may be easier for users to use a variety of existing monitors. We also believe that the use of standards improves the extensibility of tools. In other words, it should become easier to add new monitoring techniques without using ad hoc solutions, and the implementation of new monitors may be simpler due to reuse of code.

A second contribution of the paper is the identification of a number of different monitoring activities that can be described by the following categories.

File access monitors are used to read and write information in files. *Simulation control monitors* are used e.g., to start and stop simulations, to determine the length of sub-simulations, or to select the order in which certain events occur. *Visualisation monitors* can be used to visualise the behaviour of the system during a simulation, e.g. by updating MSCs or domain-specific graphics. *Performance monitors* measure and report on the performance of the system. Communication between a simulator and an external process can be controlled via *communication monitors*. Finally, *property monitors* can be used to do functional analysis of a model. Functional analysis is concerned with proving that the system behaves as expected or that certain state or event properties hold for the system. In Design/CPN it is possible to construct all of these different kinds of monitors using the monitoring framework.

The third contribution of the paper is to illustrate how monitors can be created for CP-nets. In a CPN tool, it should be possible for a user to select parts of a model and then have the tool automatically deduce which elements represent the state of the system and which elements represent events in the system. Using the information selected by a user, it is possible for the tool to generate much of the code that is required for creating a monitor that satisfies the proposed interface for monitors. Many monitors can be constructed so that they are (relatively) independent of the specific model. That makes it possible to integrate so-called *standard monitors* into the tool. A user only needs to specify minor parts of a standard monitor – the rest of the monitor is predefined. Examples of standard monitors are a data collection monitor that calculates the average number of tokens on a given place or a log-file monitor that makes it possible to write in a file each time a specific transition occurs. We also recommend that a tool should support a variety of standard monitors with varying degrees of flexibility. In other cases, it may be useful to be able to create new types of monitors or monitors that are completely model-dependent. Therefore, it is also useful that the user can create monitors from scratch. If users create monitors that conform to the monitor interface, then it should be relatively easy for tool developers to incorporate the new monitors into the tool or to create libraries of monitors that can be shared among users. If both standard and user-defined monitors are supported, the monitoring facilities should be both easy to use and still very general.

The framework is presented in general terms that are not specific to CP-nets. Therefore, the framework may serve as a reference for implementing different types of monitors for other kinds discrete-event system (DES) simulation models. The motivation for expressing the framework in non-CPN specific terms came from the realisation that many of the monitoring-related issues, that are discussed in Sect. 3.1, are probably also faced by developers of other DES simulator. The description of the basic tasks of a CPN simulator from Sect. 3.1 describe the basic tasks for DES simulators if *marking* is replaced by *state* and *transition* is replaced by *event*. Many DES simulators already contain monitoring facilities similar to what is discussed above, such as support for: updating domain-specific graphics, collecting data, and saving information in files.

The framework presented in this paper has a number of strengths and weaknesses. The framework supports the philosophy that there should be a clear

distinction between modelling the behaviour of a system and monitoring the behaviour of the model. We have shown that it is possible to use the framework to create monitors for Design/CPN. The use of these monitors should promote the creation of CPN models that are cleaner and more understandable than some of the CPN models that have been created in the past. By standardising the interface to monitors it should be easier for users to learn to use a variety of different kinds of monitors, and it should be easier for tool developers to implement new monitors, including monitors that are suggested by users. The monitoring framework is presented in very general terms and can serve as inspiration and a reference for implementing different types of monitors for discrete-event system modelling tools.

The major weakness of the paper is reflected in the title: it is a proposal for working *towards* a monitoring framework for DES simulators. No studies have been done to see if the framework is actually applicable when developing other kinds of DES simulators. The framework would probably also have wider appeal if it were expressed in an object-oriented language rather than SML. Furthermore, I am currently the only user of the few monitors that have been created for Design/CPN. We will only be able to evaluate the ease-of-use and practicality of the monitors when monitors are fully integrated into a CPN tool and are released for general use.

3.3 Related Work

Monitoring is a term that has been introduced to provide a generic term which covers a number of related, yet distinctly different, facilities in CPN simulators. We are unaware of any similar concepts. This section discusses related work regarding different approaches to monitoring simulations, and specific examples of performance-related monitors in other PN tools.

3.3.1 General Monitoring Techniques

Many simulation tools support monitoring capabilities that are not directly related to modelling and simulating the behaviour of systems. Support for accessing the monitoring functionality is provided in several different ways. Some monitors are seamlessly incorporated in predefined modules, while other monitors must be hard-coded by a user directly in a model.

Module-Specific Monitors Many simulation tools provide libraries of predefined, or template, modules that can be combined to create models. Examples of standard modules are queues, resources, and workload generators. Predefined monitors can be associated with predefined modules. For example, most tools that have queue modules can automatically calculate average queue length and average queue delay. Arena provides standard monitoring facilities for many standard modules. By activating a couple of check boxes in Arena, a user can activate monitoring facilities that will measure the performance of a queue and that will animate the state of the queue during a simulation. The benefits of

predefined monitors are that they are very easy to use, they generally monitor the most relevant aspects of the corresponding module, and they do not affect the behaviour of a model.

Augmenting Models If predefined monitors are not available, then it may be possible to augment models in order to access available monitoring facilities. Ex*Spect* supports several monitoring activities such as data collection and animation, which will be discussed in more detail below. All monitoring facilities are only able to monitor the markings of places, and each monitor can monitor only one place. This means that all relevant information must be encoded in token values, and some monitoring facilities require that a place has a particular colour set. As a result, many models created in Ex*Spect* will contain extra places and transitions that do not model the behaviour of the system in question. On the other hand, the monitoring facilities in Ex*Spect* are easy-to-use, flexible, and general.

It may not be necessary to add extra nodes or modules to a model in order to access monitoring facilities. In some cases it may be possible to exploit the modelling language or to use workarounds in the modelling tool in order to monitor a simulation. For example, many of the net inscriptions in CP-nets may be arbitrarily complex, and if the inscription language is fairly expressive, then it should be possible to encode monitoring facilities directly in net inscriptions. An excellent example of a tool in which there is ample opportunity to exploit the modelling language and the tool is the Renew [109] tool for reference nets [81] which are high-level nets. The inscription language for Renew is Java [45], and arbitrary Java methods can be called when evaluating transition inscriptions. As a result, the Renew tool implicitly supports many monitoring activities, since all of Java's functionality can be accessed from net inscriptions.

Related Frameworks It has been suggested that the monitoring framework resembles the *Observer pattern*, which addresses related issues in the area of software development. Even though I am not terribly familiar with patterns, I would like to address this suggestion because it seems like a reasonable claim. A quick search on the WWW can give a general idea about the nature and purpose of patterns and the Observer pattern. According to Appelton [4], a pattern can be defined thusly:

> "... a pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it also explains why the solution is needed!"

Based on this definition, I would be inclined to agree that the monitoring framework resembles a pattern. The monitoring framework proposes a solution to the problem of observing and extracting information from the states and events of CPN simulations, and we have also discussed why we need a solution to the problem.

The Observer pattern addresses the specific problem of keeping multiple views of a single object in sync with the object [99], according to the web site for

Object Oriented Tips. Furthermore, the web site includes the following quote regarding the Observer pattern from the highly esteemed *Design Patterns* [50]:

> "The Observer pattern describes how to establish these relationships. The key objects in this patterns are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state."

Clearly, there is a similarity between the Observer pattern and the monitoring framework. Monitors are comparable to observers, and the state of a CPN model could certainly be a subject. What is not so obvious, however, is how the events, i.e. the occurring transitions, in a CPN simulation fit into the equation. What would it mean for a transition to "undergo a change in state"? While the monitoring framework coincides with the Observer pattern on some levels, the monitoring framework and the Observer pattern do not address the same problem. The monitoring framework addresses the problem of extracting information from different objects, while the Observer pattern addresses the problem of synchronising different views of a single object. Is it reasonable to apply the Observer pattern to solve a problem that it is not intended to address? Furthermore, it is unclear whether an observer may observe more than one subject similar to the way in which a monitor can observe several places and transitions in a CP-net. At the moment I am unable to answer these questions, due to my limited knowledge of patterns. However, the comparison of the monitoring framework to the Observer pattern is certainly worth further investigation.

3.3.2 Data Collection

There are many ways in which a simulation can be monitored in a performance study. Extracting data during a simulation is the most obvious example of performance-related monitoring. By necessity, all Petri net tools that support performance analysis must monitor simulations in order to collect data. In low-level PNs, places contain a number of indistinguishable tokens. In high-level PNs, tokens carry arbitrarily complex data values. Consequently, distinctly different kinds of data can be extracted from simulations for low-level and high-level PNs.

Low-Level Petri Nets Data collection monitoring is never reflected in low-level PN models; it is always implemented directly in the tools. Most of the tools that support low-level PNs can automatically calculate some performance measures, such as average number of tokens on a place, token distributions, or transition throughput. User-defined performance measures can also be calculated by many of these tools through the use of *reward functions* [10]. Reward functions are defined from markings to real numbers. Standard formulas can be used to calculate, e.g., the average reward during a simulation or the probability that a certain condition of the net is true, In the tools, simple expressions

can be used to calculate the expected number of tokens on a place. For example, E(#P2) will calculate the expected number of tokens, i.e. the average number of tokens, on place P2. Similarly, the probability that the marking of place P2 will be less than 3 can be expressed as P(E(#P2)<3). Grammars are provided for constructing linear combinations of these kinds of expressions, as a result it is possible to define more model-dependent performance measures. Compared to the `check` and `observe` functions[1] that need to be defined for data collection monitors, such expressions have the considerable advantage of simplicity – they are much easier to define and understand. Furthermore, these expressions can be used to calculate performance measures using simulation or analytic methods.

In UltraSAN and its successor Möbius [33], a performance measure can be calculated by monitoring both places and transitions[2]. A performance measure is defined by a *reward structure* [111] consisting of two kind of reward functions: a *rate reward* and an *impulse reward*. A rate reward is similar to a reward function, as defined in [10]. An impulse reward is defined from transitions to real numbers. The reward calculated by a particular reward structure is the sum of the rate reward and impulse reward that is calculated. Each rate reward is defined as a set of predicate/function pairs that are similar to the `check` and `observe` functions for data collection monitors. The predicates define the conditions in which the rate reward return user-defined values. In all other situations the rate reward returns zero.

Reward structures are similar to the data collection facilities in data collection monitors. One major difference is that the `check` and `observe` functions can be defined for subsets of reachable markings and occurring transitions. Rate and impulse rewards must be defined for all reachable markings and all transitions because it must be possible to calculate a reward for a reward structure using either simulation or analytic models. Users must be aware that impulse functions can only be defined for timed transitions, i.e. it is not possible to calculate performance measures that are dependent on untimed transitions.

High-Level Petri Nets The potential exists for collecting different kinds of data from high-level PNs. When examining markings, it is possible to extract a data value based on the values of the tokens on places. For example, recall that in the stop-and-wait model, the queue of packets waiting to be sent is modelled as a list, and the length of the queue corresponds to the length of the list. Similarly, it may be possible to examine the data values of the tokens that are added and removed when a transition occurs during a simulation. In a CP-net, the arc inscriptions can contain variables. When a transition occurs, all variables from the surrounding arcs are bound to specific values that correspond to data values for the tokens that are added and removed. In Design/CPN it is possible to extract information from the binding of the variables of each transition that occurs.

[1] The `check` and `observe` functions for data collection monitors are analogous to predicate and observation functions for data collectors in the Performance Tool.

[2] In SAN terminology transitions are actually called *activities*.

Performance measures for SWNs can be calculated in GreatSPN, which is the most advanced PN performance tool for high-level PNs. For many years, it was only possible to calculate simple performance measures, such as average number of tokens on a place or average transition throughput. With the most recent release[3] of GreatSPN, it is possible to calculate *refined performance measures*. For a place this means that it is possible to calculate three kinds of measures: average number of tokens regardless of colour, average number of tokens for each colour in the colour set of the place, and average number of tokens that satisfy criteria based on the colour set. Similar measures can be defined for transitions. As for GSPNs, it also is possible to calculate the probability that a certain logical condition is satisfied, and to combine estimates of averages and probabilities in order to create more model-dependent performance measures. Even though it is now possible to define performance measures based on the average number of particular kinds of tokens on a place, it is apparently still not possible to define performance measures that are based on data extracted from token values.

In the Ex*Spect* tool, all performance-related data can be viewed during a simulation using *dashboard objects*. Dashboard objects are graphical objects that can be used to examine and change the contents of places during a simulation. For example, a graphical representation of a thermometer can display the number of tokens on a place, and a text box can display the token value of the token that was most recently added to a place. There are many different kinds of dashboard objects for displaying performance-related data, but, as mentioned previously, they require that all performance-related data is encoded into token values on one place. Performance measures can not be calculated by directly observing transition occurrences. In order to calculate performance measures that are related to transitions, such as the number of times a particular transition occurs, it is necessary to incorporate the relevant information into a place that is connected to the transition. The newly created data collection monitors can extract data from both markings and bindings of occurring transitions.

3.3.3 Additional Performance-Related Monitors

In addition to collecting data, there are a number of other kinds of monitoring facilities that are useful in performance studies. This section discusses a number examples of such monitors, including simulation breakpoint monitors which were introduced in Sect. 2.2.

Simulation Breakpoints It must be possible to define when a terminating simulation should stop. The most common method for stopping a terminating simulation is to stop it after a certain amount of model time has passed. This is the most common method because it is often the only method that is available in a simulator, and this is true for the PN performance tools. More precise stopping criteria can be defined if it is possible to monitor a simulation. Ex*Spect* is one of the few PN tools that support advanced stopping criteria. In Ex*Spect*

[3]GreatSPN2.0.2 was released in October 2001.

it is possible to indicate that a simulation must stop when the contents of a place changes. Defining such stop criteria is easy – the user clicks on a checkbox in a dialog box, but, again, the contents of the place must model the condition for stopping a simulation. Simulation breakpoint monitors are more flexible since they can examine a number of places and transitions, but they are more difficult to define because a user must write an appropriate `check` function for the monitor.

File Access Being able to access files during a simulation is also extremely useful. There are several advantages to using input files for generating workload for a high-level PN model. These files can contain large amounts of domain-specific information that may be difficult to include directly in net inscriptions in a model. If input files are used, it becomes possible to use the same workload for a model and the system being modelled, which is particularly useful when calibrating the behaviour of the model with the behaviour of the system. Furthermore, using the same workload for several different models, may have the same effect as using CRN for reducing the variance in performance measures. In Ex*Spect* it is possible to use files to create initial markings for places. Renew supports the use of input files by virtue of the use of Java as the inscription language. These issues are not relevant for low-level PNs since workload for low-level PNs are indistinguishable tokens.

A variety of useful kinds of information can also be saved in files during simulations. Individual data values (rewards) that are observed can be saved and used to visualise the behaviour of a model during one simulation, or the data can be post-processed, e.g. to find the proper length of the warmup period. The reward that are calculated in tools for low-level PNs are only only used to calculate measures, but the data collected by monitors can also be saved in files. With Ex*Spect* it is also possible to export the history of the contents of a place. When a token is added to a place, its token value can be saved in a file or exported directly to an executable file (.exe) file. Workload that is generated in one model can also be saved during a simulation and then later imported into another model, as mentioned above.

Domain-Specific Animation Many CPN projects [24, 107, 91] have shown that it can be extremely beneficial to use domain-specific graphics to animate the behaviour of a CP-net during a simulation. Few PN tools support monitoring techniques for updating such graphics during simulations. The Ex*Spect* tool is one notable exception. Both standard, e.g. MSC, and user-defined dashboard objects can be used to animate and visualise the behaviour of a CPN model during simulation. Dashboard objects are easy to use as there are several predefined dashboard objects, and the tool provides excellent support for creating user-defined dashboard objects. In contrast, the Mimic library [106] for Design/CPN is difficult to use for inexperienced users, and there are currently no libraries of graphical objects. Since all Java functionality can be accessed in transition inscriptions in Renew, the Renew tool also implicitly supports the use of domain-specific graphics.

Chapter 4

Annotating Coloured Petri Nets

annotation (*noun*) a note added by way of comment or explanation

- Merriam-Webster's Collegiate Dictionary

This chapter discusses the paper "Annotating Coloured Petri Nets." Section 4.1 introduces the concept of annotations. Section 4.2 summarises the main contributions of the paper and evaluates the techniques that were applied. Section 4.3 discusses related work.

4.1 Introduction and Background

CP-nets can be used for several fundamentally different purposes such as functional analysis, visualisation, and performance analysis. It is seldom the case that the exact same CP-net can be used for a variety of different purposes, as it is frequently necessary to make small or large modifications to a CP-net in order to obtain a CP-net that is appropriate for another purpose. These modifications include adding places and transitions, adding code segments, and modifying colour sets and arc inscriptions. There are a number of disadvantages associated with modifying a model. Making the modifications can be time consuming and error-prone. More importantly, there is no guarantee that the behaviour of the model will not be affected by the modifications that are made.

Up to this point it has only been partially possible to use a CPN model for different purposes without having to change the CPN model itself. With the current tools, it is possible to do, e.g. performance analysis without adding transitions and places for the sole purpose of doing the performance analysis. The monitors described in Chapter 3 also support a separation between modelling the behaviour of a system and monitoring the behaviour of a model. In the future, it may be possible to use monitors rather than code segments or extra places and transitions to access auxiliary facilities, such as the libraries for generating MSCs or for updating domain-specific graphics. Unfortunately however, it is often necessary to add extra information to colour sets and arc inscriptions to hold, e.g. history information for the objects modelled by the tokens. An example of such history information is the arrival time for a packet in the stop-and-wait model.

There are many situations in which it would be useful to augment some token values in a CP-net with information that is not related to modelling the behaviour of the system in question. We use the term *annotations* to refer to pieces of information that are not necessary for determining the behaviour of a model. The arrival time of a packet is an example of an annotation that is useful in the context of performance analysis, assuming that the behaviour of the model does not depend on the arrival time of a packet.

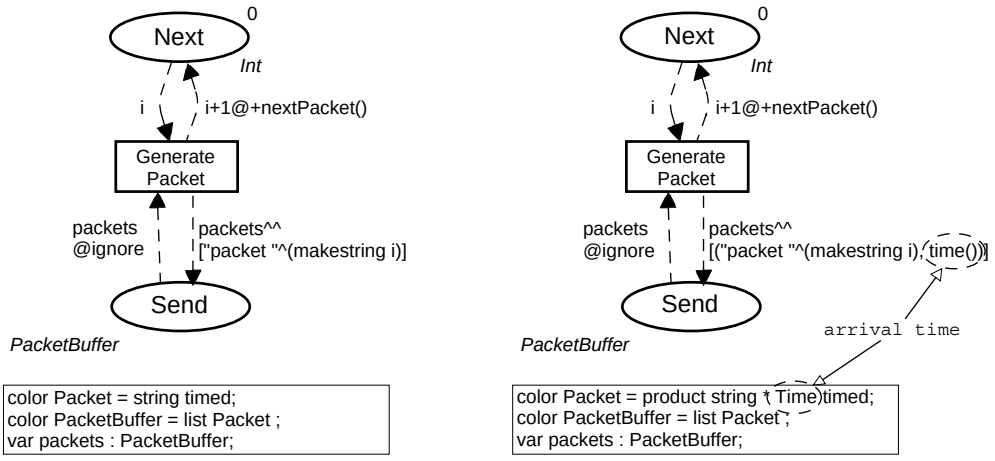
Annotations are particularly useful when analysing the performance of CPN models. In order to calculate performance measures for a CPN model, data must be collected from the model during a simulation. Section 1.3 describes how the average length of the queue of packets waiting to be sent in the stop-and-wait model can be calculated. The length of the queue, i.e. the length of a list which is a token value, is measured periodically, and the length of the queue is used to calculate the average length of the queue. This particular performance measure can be calculated by extracting data from information that is naturally available in the model.

Other kinds of performance measures can be calculated if additional information is introduced into the model. Tokens in CP-nets can carry complex token values. Consequently, it should be possible to measure packet delay, i.e. the amount of model time that passes from when a packet it added to the queue at the sender until the packet is successfully received by the receiver. Figure 4.1 shows how the stop-and-wait model should be modified in order to measure packet delay. Figure 4.1.(a) shows the Upper Layers Send module in which packets are generated and added to the queue. The declaration of the data type for a packet (`Packet`) can be seen in the box beneath the model. A packet is a string. The easiest way to measure packet delay is to augment the `Packet` data type with additional information, so that it contains the time at which the packet is added to the queue. Figure 4.1.(b) shows the modifications that need to be made in order to incorporate this auxiliary information into the Upper Layers Send module. The data type for a packet is changed to be a pair where the first element is a string and the second element is of type `Time`. The arc inscription on the arc from Send to Generate Packets must also be updated to include the time at which a packet is generated (the function call `time()` is assumed to return the current model time). At least one arc inscription in the Receiver module will also have to be changed in order to easily inspect the arrival time of packets when they are received.

It is quite useful to add this kind of auxiliary information to a CP-net for the purposes of performance analysis. However, in most cases it will not be as easy as it was for this example. In more realistically sized models, more auxiliary information will be added to tokens, more declarations will have to be changed, and more arc inscriptions will have to be modified.

4.2 Main Contributions

Many CPN users are familiar with the problem of maintaining several slightly different versions of a CP-net in order to analyse different aspects of a system. In



(a) (b)

Figure 4.1: Generating and adding packets to the queue. (a) Packet arrival time ignored. (b) Arrival time included in packet.

many cases, extra information needs to be added to (or removed from) colour sets in a CP-net in order to obtain a CP-net that is appropriate for a given purpose. This is true even for very simple CP-nets — consider, for example, the resource allocation system that is found in Jensen's volumes on CP-nets [70, 71]. In the system there are two kinds of processes (`p` and `q`) which allocate and deallocate different amounts of `R`, `S` and `T` resources. At least three variations of the resource allocation CP-net can be found in these volumes. Figures 4.2.(a) and 4.2.(b) show two of the three variations: part (a) shows the *basic* CP-net which models the basic aspects of the system, and part (b) shows the *extended* CP-net which includes cycle counters for the `p` and `q` processes. The net structure is the same for the two CP-nets, but many of the colour sets and net inscriptions in the basic CP-net have been manually modified in order to obtain the extended CP-net. The cycle counters are a good example of annotations: they represent auxiliary information that is useful for analysing the system, but they do not determine the behaviour of the system, nor do they represent any aspect of the system being modelled. It can be shown using complicated proofs (see [71] for details) that the behaviour of the two CP-nets is very similar – essentially, every reachable marking of the CP-net in Fig. 4.2.(b) is the same as a reachable marking of the CP-net in Fig. 4.2.(a) if the cycle counters are removed.

The main contribution of this paper is a method for augmenting tokens in a CP-net with auxiliary information that affects the behaviour of the CP-net in a very limited and predictable manner. With this method, annotations are not integrated into colour sets and arc inscriptions in a CP-net; rather, they are defined separately from the CP-net in so-called *annotation layers*. An annotation layer defines annotations and how the annotations are to be associated with the tokens in a particular CP-net. An annotation layer is always

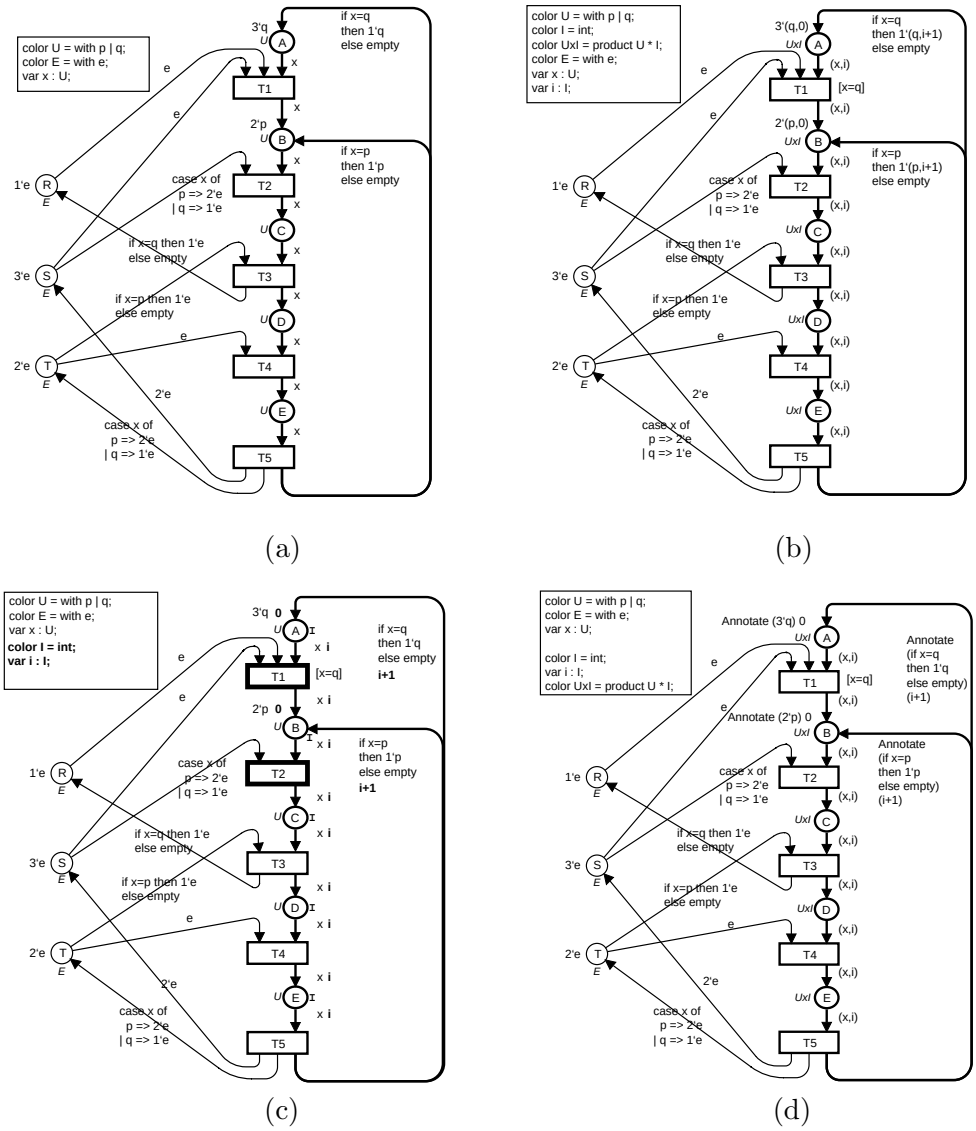


Figure 4.2: Resource allocation system. (a) Basic CP-net. (b) Extended CP-net with counters. (c) Annotated CP-net with counters defined in annotation layer. (d) Matching CP-net with counters.

defined for one particular CP-net which is called the *underlying CP-net* of the annotation layer. An *annotated CP-net* is a pair consisting of an annotation layer and its underlying CP-net. Figure 4.2.(c) shows an annotated CP-net for the resource allocation system. The underlying CP-net is shown in grey (it is the same as the CP-net in Fig. 4.2.(a)). The annotation layer is shown in black. The annotation layer contains *auxiliary declarations* and *auxiliary net inscriptions*, such as *auxiliary colour sets* and *auxiliary arc inscriptions* that are similar to their counterparts in CP-nets. Places with auxiliary colour sets are called *annotated places*. In Fig. 4.2.(c), places `A`-`E` are annotated places, each with auxiliary colour set `I`. Informally, in an annotated CP-net every

token carries a token colour, and some tokens carry both a token colour and an annotation. A token that carries both a colour and an annotation is called an *annotated token*. Just like a token colour, an annotation may be arbitrarily complex. Annotated places are the only places that can contain annotated tokens.

Rather that defining semantics for annotated CP-nets, we define a translation from an annotated CP-net to a regular CP-net, called the *matching CP-net*, where the annotations become an integrated part of the matching CP-net. Figure 4.2.(d) shows the matching CP-net for the annotated CP-net in Fig. 4.2.(c). The translation is straightforward: the net structure for the matching CP-net is the same as the original CP-net, and the net inscriptions from the original CP-net and the annotation layer are combined to form net inscriptions in the matching CP-net. Translating an annotated CP-net to a matching CP-net must be straightforward because it must implemented in a CPN tool. A marking of a matching CP-net is said to *cover* a marking of its underlying CP-net, if the two markings are equal when annotations are ignored in the first marking. A binding[1] of a transition in a matching CP-net is also said to *cover* a binding of the corresponding transition in the underlying CP-net, if the bindings are equal for all of the variables of the transition in the underlying CP-net. The set of variables of a transition in a matching CP-net always contains the set of the variables from the corresponding transition in the underlying CP-net. Similarly, a step[2] in a matching CP-net can be said to cover a step in its underlying CP-net.

When certain conditions are fulfilled, we can guarantee that adding annotations will affect the behaviour of the underlying CP-net in a very limited and predictable way. For example, one requirement is that every auxiliary arc expression must evaluate to a single annotation, in contrast to regular arc expressions which must evaluate to multi-sets of colours. If a matching CP-net is derived from an annotated CP-net with a so-called *sound* annotation layer, then the following properties hold. Every marking of the matching CP-net covers a marking in its underlying CP-net, and every step in the matching CP-net covers a step in the underlying CP-net. Furthermore, every marking in the underlying CP-net, can be covered by a marking in the matching CP-net, and every step in the underlying CP-net can be covered by a step in the matching CP-net. The initial marking of the matching CP-net covers the initial marking of its underlying CP-net. An occurrence sequence in the matching CP-net covers an occurrence sequence in its underlying CP-net. Similarly, every occurrence sequence in the underlying CP-net, can be covered by at least one occurrence sequence in the matching CP-net. When these properties hold, the behaviour of the matching CP-net is said to *match* the behaviour of its underlying CP-net.

Defining annotations in layers makes it possible to make modular definitions of both a CP-net and one or more layers of auxiliary information that can be used for various purposes. Since annotations are defined separately from

[1] A *binding* of a transition t is a function that assigns a proper value to each of the variables of t, and the guard of t must evaluate to true for the given values (Def. 2.6 in [70]).

[2] A *step* is a multi-set of *binding elements*, where a binding element is a pair (t,b) consisting of a transition t and a binding of t (Def. 2.7 in [70]).

a CP-net, it easy to disable annotations if a CP-net is to be used for another purpose. By defining several different layers of annotations, it is possible to maintain several versions of a CP-net and thereby to use the same basic CP-net for various purposes by adding, removing, or combining annotation layers.

This paper presents a potentially useful method for augmenting tokens in a CP-net with auxiliary information. There are several advantages to the method. The method supports the separation of modelling the behaviour of a system and monitoring the behaviour of the model, since non-essential information is not included in the colours of the tokens in the underlying CP-net. Furthermore, it is shown that when certain conditions are fulfilled, the annotations that are added to a CP-net will affect the behaviour of the CP-net in a very limited and predictable manner. No new semantics need to be defined for annotated CP-nets, since an annotated CP-net can be automatically translated to a regular CP-net. Several annotation layers can be defined for a CP-net for a variety of different purposes, such as data collection, updating MSCs, or communication with other processes, and it should be easy to enable and disable one or more layers of auxiliary information. Annotations should be particularly useful when using CP-nets for performance analysis.

The major drawback for the proposed method is that it has not yet been used in practice. Clearly, it is important that support for annotations be implemented in a CPN tool in order to investigate the practicality and usefulness of the proposed method. In addition, some of the definitions and requirements that are presented in the paper are unnecessarily restrictive. For example, it is only possible to add annotations to a CP-net if all arcs from annotated places to transitions have arc expressions that evaluate to a single colour, and it is only possible to add one particular annotation to a multi-set of colours. Many of the requirements and restrictions are included because we wanted to define the annotation rules so that they are straightforward to implement, to use and to understand. Practical experience with the method may show us how to weaken or remove the restrictions.

4.3 Related Work

It is frequently helpful to add auxiliary information to models in order to analyse different aspects of a system. However, it is not always clear what impact the auxiliary information has on the behaviour of the model. This section discusses alternative methods for adding auxiliary information to models, methods for comparing the behaviour of different models, and the use of formal methods for performance analysis.

4.3.1 Adding Auxiliary Information

Models can contain two kinds of information: information that is necessary for modelling the behaviour of the system, and auxiliary information that does not influence the behaviour of the model. In some cases, it is possible to analyse additional aspects of a system when auxiliary information can be added to

models. We have seen how annotations can be used in CP-nets to measure, e.g. average packet delay. This is possible only because relevant information can be associated with each token. Measuring average packet delay is more complicated when using low-level Petri nets or SANs because the tokens cannot carry any information. For example, average packet delay could be measured using a technique similar to what is used by Sanders and Meyers [111] to measure average response time for jobs in a multiprocessor system. Average response time must be calculated as the number of jobs in the system divided by the average arrival rate of jobs. This requires a detailed understanding both of the model and of relevant statistical formulas.

In high-level PNs tokens can carry information. In Ex*Spect* all auxiliary information must be hard-coded in token colours, in contrast to defining annotations separate from a CP-net. However, a user can use standard building blocks and predefined colour sets which contain many useful kinds of auxiliary information. For example, the `object` colour set has components for arrival time, processing time, and object type, which can be updated and monitored at appropriate times during a simulation. The `generator` building block generates tokens with `object` data values, and it will, e.g. add the appropriate arrival time to a token colour when a particular transition occurs.

General simulation packages often have a notion of *entities*, where entities represent, among other things, objects that move about in the system being modelled. Entities could be used to model, e.g. packets in a computer network, customers in a bank, or cars in an assembly plant. Auxiliary information can be associated with entities using *attributes* which contain information that is specific for individual entities. Attributes can contain information that determine the behaviour of the model, such as a customer type that determines how a customer is added to a priority queue. Attributes can also contain auxiliary information which does not influence the behaviour of the system but which does aid in analysing the behaviour of the model. An example of this kind of attribute is the arrival time of a customer.

Incorporating auxiliary information in a model generally requires either manually encoding the information in the model or building the model with predefined modules and data types. The advantages of using standard modules and data types is that they are easy to use and they generally contain the information that is relevant for most situations. Problems arise, however, when a user is confronted with a new kind of system which cannot be properly modelled by the standard modules and data types. Models containing auxiliary information will generally be more complicated than models that do not contain auxiliary information. As a result, the models with auxiliary information may be more difficult to understand and to validate. With annotations, auxiliary information can be added to a model without modifying the model and without affecting the behaviour of the model.

4.3.2 Comparing Behaviour

Work that is closely related to our notion of annotations can be found in Lakos' work on abstraction [83]. He defines a so-called colour refinement within the

context of behaviour-respecting abstractions of CP-nets. This colour refinement is used to specify more detailed behaviour in sub-modules by extending colour sets to larger domains. The refined colours are only visible in the sub-modules, and the refined colours will typically contain information that is necessary for modelling the behaviour of the system in question. This colour refinement corresponds somewhat to our way of extending colour sets by adding annotations to colours, and it respects the behaviour of the CP-net. I am not aware of work other than our own that addresses the issue of introducing *auxiliary* information into a CP-net (or any other type of simulation model) while at the same time preserving the behaviour of the CP-net. Nor do I know of any other method that can be used to automatically enable or disable auxiliary information when analysing different aspects of one particular model.

The Renew tool supports a relaxed syntax [81] for defining reference nets that allows the addition of auxiliary information to a model without affecting the behaviour of the model. The tool allows the use of untyped places and untyped variables. In other words, a place can contain tokens with token values from many different data types, and arcs can be used to add and remove any kind of token from these places. Adding auxiliary information to the initial marking of untyped places will not affect the behaviour of the model, assuming that the auxiliary information never needs to be attached to a token on a typed place. When a transition occurs, the auxiliary information can be accessed via the binding of an untyped variable in transition inscriptions. The tool requires that all variables on arcs from transitions to typed places must be typed, and if one variable is typed then all variables in the model must be typed.

It is often useful to know that two different models have equivalent, or at least very similar behaviour. When analysing the functionality of a system, an analyst may develop and validate a fairly detailed model which captures many aspects of the system. For example, a model of an application layer protocol, such as HTTP, may contain a detailed description of lower-level protocols such as TCP or IP. When the time comes to analyse the behaviour of the model, the model may be too detailed, and the desired analysis methods cannot be employed, e.g. full state space analysis cannot be used because the state space is infinite. It is then up to the user either to try to reduce the complexity of the model in such a way that the behaviour of the model is preserved, or to find alternative analysis methods that can handle the problem. For example, when modelling and analysing HTTP, it should not be necessary to include a detailed description of TCP, in which case the parts of the model representing TCP could be simplified in an attempt to obtain a smaller state space. In some cases, a user must use elaborate proofs to show that the results obtained by analysing the reduced model or by using an alternative analysis technique are, in fact, applicable for the original model.

Several techniques exist for comparing the behaviour of different models and for comparing different behavioural representations of the same model. A common technique for testing the behavioural similarities of two process algebraic models is the use of *bisimularities* [22], where a *(weak) strong bisimulation* is a binary relation that shows that two process algebraic models have (observably) identical behaviour. It must be possible to investigate the complete behaviour

of each model in order to show that two models are bisimilar. There are also rules for reducing low-level Petri nets [20] and for reducing CP-nets [58] to nets with similar behaviour. An alternative to reducing a model itself is to try to reduce the size of the state space of the model in order to analyse the behaviour of the model. State spaces with equivalence classes [70] is a technique that can be used to group similar markings and binding elements together in order to generate a representation of the behaviour of the model that is smaller than the full state space of the model. The techniques mentioned above require reducing either a model or the representation of the behaviour of the model in order to obtain a smaller, alternative representation with similar behaviour. In contrast to these techniques, we have shown that *adding* annotations to a CP-net will result in a CP-net which behaves very similarly to the original CP-net. I am not aware of any other techniques for augmenting a model that preserves the behaviour of the model.

4.3.3 Performance Analysis and Formal Methods

A large amount of research is concerned with using formal methods to analyse the performance of systems, and a recent summer school [23] provided an introduction to the formal methods that are most commonly used for performance analysis. Formal methods are characterised by Clarke et al. [34] as mathematically-based languages, techniques and tools for specifying and verifying systems, and formal specification uses a language with a mathematically defined syntax and semantics. The formal methods that are most commonly used for performance analysis are (G)SPNs, SWNs, SANs, the Performance Evaluation Process Algebra (PEPA) language [62], and queueing networks [78]. The PEPA Workbench [102, 52] supports the PEPA language, and Möbius [33] is an innovative, multi-paradigm modelling tool that supports the creation and analysis of models consisting of components that are expressed in different formalisms, such as SAN, PEPA, and SPN.

Many of the formal methods that are used for performance analysis were derived from formal methods that originally were only used for functional analysis of systems. Formal methods, such as Petri nets, activity nets, and process algebras, are used both to specify and verify system behaviour. Timing information was added to these formal methods in order to support modelling and analysis of both functionality and performance of systems. Many of the resulting formal methods, such as (G)SPNs, SANs, and stochastic process algebras, require that time delays are, for example, all exponentially distributed. When these requirements are fulfilled, it is possible to derive an analytical model, such as a continuous-time Markov chain, from the original model.

There are a number advantages associated with using formal methods for performance analysis. One major advantage is the fact that they can describe the behaviour of a system in an unambiguous way. Moreover, the same model can potentially be used to analyse both the functionality and the performance of a system, since an untimed model can be created for functional analysis, timing information can be added, and the performance of the model can then be analysed. Markovian models, which are also a formal methods, can provide

exact values for relevant performance measures, and advanced techniques exist for solving such models. However, it is often to difficult to manually define accurate Markovian models, therefore it is also advantageous that Markovian models can be automatically generated from higher-level specifications, such as SANs and GSPNs.

There are also a number of disadvantages to using formal methods for performance analysis. In 1996, Clarke observed that some of the problems with formal methods are: the notations were too obscure, the techniques did not scale, and the tool support was inadequate or too hard to use. On a positive note, Petri nets are becoming more widely accepted, case studies have shown that formal methods can be used to analyse the performance of industrial-sized systems (see Sect. 5.3.3), and the most widely used PN performance tools are quite stable. While there have been improvements in these areas during the past decade, these problems still exist. For example, textual PEPA models are not easy to understand, industrial-sized models can rarely be analysed using Markovian models due to the state explosion problem, and tools that are developed in academic research groups frequently use state-of-the-art analysis techniques, but they are sometimes difficult to use, poorly documented, and not terribly robust. With respect to these problems, commercial products that are based on informal modelling languages are likely to be more popular (and useful) than formal methods for several reasons: industrial-sized models can be built using predefined, drag-and-drop modules of familiar components; every syntactically-correct model can be simulated; customer support is provided for mature, stable, and well-documented tools; and for the average user it is still fairly time consuming to learn to understand a formal method.

Chapter 5

Case Study: Analysis of Web Servers

This chapter discusses the paper "Simulation Based Performance Analysis of Web Servers." Section 5.1 introduces the CAPLAN project in which the work presented in this paper was done. Section 5.2 summarises the main contributions of the paper and evaluates the techniques that were applied. Section 5.3 discusses related work.

5.1 Introduction and Background

The CAPLAN project [26] was a collaborative project between the CPN group at the University of Aarhus, the Hewlett-Packard Corporation, and the Danish National Centre for IT Research. The project ran from January 1998 until January 1999. The purpose of the CAPLAN project was to investigate the use of CP-nets for model-based capacity planning of distributed computing environments. The CAPLAN project developed CPN models of three distributed applications selected by Hewlett-Packard. This served as a basis for evaluating the usefulness of CP-nets and Design/CPN for capacity modelling of distributed systems. The CAPLAN project was judged to be successful, and the collaboration between the three parties continued as the HP-CPN Centre [65].

This paper presents the results from the first subproject of CAPLAN which was concerned with capacity planning and performance analysis of a web server environment. Two goals of this subproject were to develop tool support for using CP-nets for performance analysis and to investigate the practicality of using CP-nets to analyse the performance of real-world systems. As student programmers, Bo Lindstrøm and I developed the Design/CPN Performance Tool in conjunction with this project.

5.2 Main Contributions

The initial phase of the CAPLAN project considered a concrete representative example of a distributed computing environment in the form of a simple web server environment. My contributions to this project and paper deal with the simulation-based performance analysis of the *CPN web model* that was constructed by other project members. My contributions are summarised here and

will be described in more detail below. This was the first project that put the Performance Tool to practical use. The CPN web model was validated and calibrated by comparing performance results from simulation with the performance which was measured in a corresponding physical environment. Using the validated model, simulation experiments were run in order to examine the effects of varying the arrival rates of requests on the performance of the web server. Additional experiments were undertaken in order to compare the performance of different configurations, e.g. faster CPU and faster disk, with the basic configuration corresponding to the actual web server.

The web server that was modelled was an Intel Pentium II 266 MHz processor based PC equipped with a local disk, and 160 MB RAM of internal memory. For the experiments the web server application was configured as a single threaded web server with cache disabled. The aim was to configure the web server and choose parameters in a way which made it possible to put heavy load on the server by means of relatively few requests. As a consequence, the configuration was not realistic and was far from optimal with respect to performance of the web server. However, this rather disabled configuration was better suited for initial efforts to calibrate the performance of a CPN model with the performance of a complex real-world system.

Workload for the actual web server was generated by a *trace client* that takes as input a web access log file and then makes a replay of the get-requests contained in the web access log file. Each get-request (entry) in a web access log contains a time stamp, specifying when the requests are to be made, and a specification of the document requested. Access log files were generated after analysing a real log file from a local web server. It was determined that the file sizes fit a Weibull distribution with an average file size of approximately 5KB. To avoid any effect related to the file cache of the web server all get-requests in the generated workload are for distinct files. Due to time constraints, a detailed analysis of neither the request times nor temporal locality of requests in the web access log file was undertaken. We assumed that the workload during peak hours was 5-10 requests/second, which is not an unreasonable expectation judging by what has been observed within other academic environments [6]. Furthermore, we assumed that the interarrival times between requests are exponentially distributed. This is not the most accurate model for request arrivals [6], but time constraints prohibited us from making a more precise analysis and representation of arrival rates.

The size and complexity of the CPN web model precluded using state space analysis to fully validate the functional correctness of the model, but interactive simulations and MSCs were used extensively to validate the logical behaviour of the model. While these techniques can never ensure that a model is error-free, they are useful for finding obvious modelling errors, and they can confirm that the model behaves as expected in a variety of situations.

The goal of the *calibration* process was to obtain a match between the performance results obtained by simulation and the performance results measured in the actual web server environment and in this way to obtain a validated CPN web model. In this phase, web access log files were used to generate the same requests for both the real web server and the CPN web model. The process of

calibrating the model parameters consisted of a number of iterations of adjusting parameters in the model, running simulations, and comparing the results to the performance of the actual web server.

Another contribution of the paper was that we illustrated the kind of performance results which can be obtained from lengthy simulations of the CPN web model. When considering the performance of a web server, several performance measures are of interest. *Resource utilization* of the CPU and local disk in the web server and of the network denotes the percentage of time in which each resource is busy processing jobs. *Response time*, i.e. the time from a client initiates a get-request by opening the TCP connection until the response is received, is also of interest. This is the delay observed from the point-of-view of a client. Response time is highly variable since it depends largely on the size of the document that has been requested.

During the performance analysis phase of the project, workload for the CPN web model was generated on-the-fly. The requests that were dynamically generated during a simulation fulfilled the same criteria as the workload that was described above, i.e., the file sizes of the requested documents were generated from the Weibull distribution, and the interarrival periods between requests were exponentially distributed.

The facilities that were discussed in Chapter 2 were used to analyse the performance of the CPN web model. Initial experiments examined the effects of varying the request arrival rates for between 5 and 50 requests/second. The new performance facilities were used to run a number of independent simulations, to collect data for each simulation, to save simulation output in an organised directory structure, and to calculate confidence intervals for the relevant performance measures. Each simulation corresponded to 30 minutes of activity in the web server environment. The average utilization of the different resources and the average response time was investigated for different workload intensities. Increasing the arrival rate of requests increased the utilization of all resources, but the increase in utilization of the network was not as large as for the resources in the web server since the capacity of the network was quite large (5 Mb/sec.). The new performance facilities were also used to simulate and compare the performance of alternative configurations of the web server. Paired-t confidence intervals were used to compare the utilization of resources in the original model configuration with utilization of resources in configurations in which the CPU was faster, the disk was faster, and caching was enabled.

Another contribution of the project was a modelling framework for distributed computing environments. The framework was developed by other project members. The framework is based on a building-block approach which divides the components of CPN models into three distinct layers: a structural layer describing clients, servers, networks, and their relationship; an application layer describing the applications running on the servers and the clients; and a resource layer describing the resources, i.e., CPUs, disks, and communication channels, of the system. This means that the CPN model includes both a functional view of the system represented by the application layer, and a performance view represented by the resource layer. Separating the parts of the model giving a functional description (the application layer) from the parts

of the model describing the use of resources implies that it is easy to make the transition from a CPN model focusing on performance to a CPN model focusing on the logical correctness of the system. This can be done by simply disabling the pages in the resource layer since all aspects of the CPN model related to performance are isolated in this layer. Of course, one must ensure that the resource layer does not affect the functionality of the application and structural layers. The framework provides no guidelines regarding this problem, and in practice it will generally be difficult and time consuming to show that the resource layer does not affect the functionality of the application layer.

This project made several useful contributions regarding the practicality of using CP-nets for performance analysis. One of the conclusions from this project is that it is relatively easy to collect data from CP-nets for performance analysis. It is no more difficult to develop CPN models for performance analysis than for other types of analysis, and defining data collectors for calculating performance measures was also relatively easy. An additional advantage of using Design/CPN was that the same workload could be used for both the actual system and the model of the system during the calibration phase of the project.

This project revealed that more sophisticated support for performance analysis using CP-nets was required, in addition to support for data collection,. As inexperienced data analysts, we experienced some of the pitfalls associated with simulation-based performance analysis: statistical analysis techniques were improperly applied, results from one simulation were taken to be the true answers, and a great deal of time was spent modelling the system and little effort was spent analysing the simulation output. Even though the modelling and initial analysis of the CPN web model was completed by May 1999, it was not until March 2001 that proper analysis of the performance of the model was undertaken using the performance facilities that are described in Chapter 2.

It may be argued that the CPN web model is simplistic: the server was configured to run with only one thread, caching was disabled, and the workload of get-requests was not realistic. However, a goal of the project was to compare the performance results obtained by simulations with the performance monitored in a corresponding physical environment. Therefore, a web server environment which could be setup and controlled within the scope of the project was important. The main goal of this project was to investigate the feasibility of using CP-nets to analyse the performance of industrial-sized systems, i.e. our main purpose was not to make detailed analysis of the performance of a web server.

5.3 Related Work

In this case study, we used a graphically-based, formal method to build and analyse the performance of a fairly detailed model of a web server. This section discusses work related to these areas. There are numerous modelling languages and tools to choose among, and a comparison and discussion of various possi-

bilities is found below. The problem of building industrial-sized models with formal methods is also considered. The section ends with surveys of industrial case studies using formal methods and case studies of web environments.

5.3.1 Modelling Languages, Formalisms and Tools

There are a multitude of languages and formalisms for modelling and analysing the performance of discrete-event systems. Some languages are text-based, while others have a graphical representation. The semantics of modelling languages vary from informal or ad hoc to formal and well-defined. Some simulation packages are designed for modelling and simulating one particular kind of system, and other packages can be used to model almost any kind of system. The advantages and disadvantages of these diverse kinds of modelling languages are discussed below.

Textual simulation languages, such as SIMNET [118], SIMAN [101], and Parsec [9], are essentially high-level programming languages with simulation-related primitives and run-time systems that are specialised for running DES simulations. These languages are very general since they can be used to model virtually any kind of discrete-event system. The main drawback to these languages is that the models can be difficult to create and understand because the syntax of the modelling language is complex and confusing. Models that are graphically-based, such as Petri nets or Arena models, are often more intuitive to both to create and to understand. This can facilitate discussions between modellers and experts from the system domain. An interactive simulation of a graphical model is also a much more intuitive debugging mechanism than text-based debuggers. A survey of simulation software [117] reveals that the majority of popular simulation packages provide support for graphical model construction.

Some of the formal methods that can be used to model and analyse the performance of systems were discussed in Sect. 4.3. The primary advantage of using formal methods is that they have well-defined semantics, and it is possible to model the behaviour of a system in an unambiguous way. The models in other simulation packages, such as Arena or SIMNET, may not have well-defined semantics, and situations may arise in a simulation in which the model behaves differently than a user expects due to the fact that the tool implements an ad hoc semantics. For example, if two events can happen at the same time, a user may expect that one of the events will be chosen non-deterministically, when in fact the simulation tool will always be predisposed to choose one event before the other. Unfortunately, this kind of problem can also be found in tools that support formal methods. For example, if two or more transitions were concurrently enabled in an older version of a high-level PN tool [7], then the transitions would occur in an order determined by the alphabetical order of their names rather than occurring concurrently or in a non-deterministic order.

The modelling languages (e.g. PNs, SANs, PEPA, and SIMAN) and simulation packages (e.g. Arena and SIMNET) that have been discussed until this point have all been general-purpose tools that can be used to model many dif-

ferent kinds of systems. These tools are very general and flexible. However, a user must spend a significant amount of time building models, validating the models, and, finally, analysing the behaviour of the models. Other tools are tailor-made for modelling and analysing one particular kind of system. For example, GloMoSim [8] is a simulator environment for ad hoc wireless networks, and Woflan [124] is a Petri net-based tool for analysing business workflows. An advantage of domain-specific tools is that they often provide libraries of validated modules representing elements in the domain area. For example, GloMoSim has models of many network protocols from several layers of the OSI network architecture. This means that models can be built fairly quickly and easily by combining and parameterising the predefined modules. Another advantage of domain-specific tools is that they generally provide good support for analysing relevant aspects of the systems in question. For example, GloMoSim automatically measures throughput and packet delay in the different layers of the network architecture, while Woflan checks, e.g. that a workflow process definition is sound, i.e. that it is always possible to complete a task. A disadvantage of using domain-specific modelling tools is that it may be difficult or impossible to create models of systems that are only slightly different from the systems and modules that the tools support.

5.3.2 Formal Methods and Industrial-Sized Models

Perhaps the biggest obstacle in using formal methods for analysing the performance of systems is that the techniques do not necessarily scale well to large, industrial-size models. This is true for both specifying and analysing the behaviour of the system. Section 2.3 discussed how the state explosion problem can prohibit the generation of analytical models which are often used in connection with formal methods. It may be possible to avoid the state explosion problem by using small and unrealistic configurations of the system, but this is not desirable if the goal of a study is to analyse a realistic configuration of an industrial-sized model.

Even specifying an industrial-sized system using formal methods may be problematic. This problem concerns the accuracy of the models that can be created. For some formal methods assumptions must often be made about the system in order to use the analytical models that are available. For example, it must often be assumed that all delays within the system are exponentially distributed if analytical models are to be derived from, e.g., GSPNs, SWNs, SANs or PEPA models. DSPNs allow both deterministic delays and exponential delays, but only one deterministic transition may be enabled in each marking. In some cases, these assumptions may not be realistic, or they may be too restrictive, and the accuracy of the calculated performance measures can be affected by the assumptions that have been made.

When using, e.g. low-level, non-hierarchical Petri nets [39] or PEPA, it can be quite difficult to create an understandable model of an industrial-sized system. Complex and complicated models can be difficult to debug and validate. Many tools for low-level PNs only support the creation of monolithic, flat models. Large examples of such models are often incomprehensible, despite the fact

that they are created using an intuitive, graphically-based language. Several formal methods have mechanisms for creating large and complex models from smaller, more understandable components. The *replicate* and *join* operators for SANs make several copies of a SAN, and connect SANs by merging places and transitions, respectively. Recently, a composition mechanism [11] has been defined for SWNs. While smaller and more understandable modules can be composed into one large model, even the developers admit that the composition tool is not easy to use, there are no methods for validating the composition, and the resulting composed models are hardly readable [47]. Structuring mechanisms exist for creating hierarchical CP-nets and hierarchical queueing Petri nets [15, 63] (Hi-QPN). With these mechanisms it is possible to create a number of related modules, create new modules from existing modules, and reuse a module in several parts of a model. Furthermore, it is possible to capture different levels of abstraction of the modelled system within one model. The CPN web model presented in this paper consists of 27 pages, 96 places, and 37 transitions. It would have been extremely difficult to create a comprehensible model of this size without some kind of decomposition or abstraction mechanism.

5.3.3 Industrial Case Studies

There are a large number of case studies of concerned with analysing the performance of industrial-sized systems. Conferences, such as the Winter Simulation Conference [128] and those sponsored by SIGMETRICS [114], contain numerous examples of performance studies in the areas of computers and telecommunications, manufacturing, transportation, health care, and military. The homepage for the Arena software package [5] contains references to many studies in which significant amounts of time and money have been saved due to the results of performance studies that are based on Arena models. It is beyond the scope of this dissertation to provide an overview of the state-of-the-art results regarding performance studies of industrial-sized systems. The discussion here will be restricted to discussing case studies in which formal methods were used to model and analyse realistically-sized systems, and to case studies regarding web servers and web traffic.

Formal Methods Formal methods have been used to study the performance of many kinds of systems. PEPA models have been used to analyse multimedia streams [21] and hierarchical wireless networks [46]. Interesting performance results were obtained in each of these studies, and sophisticated techniques were used to exploit symmetries in a model in order to reduce the state space needed for generating the necessary Markovian model. However, neither of the models were terribly understandable people who are not familiar with process algebras, and there is no discussion of how the behaviour of the model was validated. Contact centers [47], and ATM switches [48] have been analysed using SWNs. It is interesting to note that the users and developers of these formalisms view simulation-based analysis as somewhat unsatisfactory: "the size of the [SWN] models will only allow simulation [...], while an interesting open problem is how to derive from these models some more compact ones to be

used for performance evaluation purposes" [18]. Admittedly, simulation cannot provide the exact results that analytical models can. However, in at least one study, the authors' fail to comment on whether it is reasonable to assume that all time delays in the system are exponentially distributed, and whether this assumption has an impact on the performance measures that were obtained.

CP-nets have previously used in other projects on performance analysis in areas such as ATM networks [35], alternative TCP implementations [38], and bank transaction processing [27]. These studies revealed that there was room for improvement in the area of using CP-nets for performance analysis. In the ATM and bank transaction studies, the modellers were responsible for defining how data was to be collected during a simulation, and in both cases all data collection functionality was hard-coded directly in the models. The bank transaction model contained an entire page that was only used for extracting data and saving the data in files. It is also unclear whether the conclusions of the ATM study are reliable. The ATM study compared the performance of four different flow control algorithms for six types of traffic, and only one simulation was used for each configuration. Since each simulation represented 3 seconds of real time, it seems unlikely that the simulations could be considered steady-state simulations, in which case it is insufficient to run only one simulation for each configuration. The performance study of the CPN web model has shown that it is easy to collect data from a CP-net during a simulation without having to incorporate data collection functionality directly in the model. Furthermore, the new performance facilities should aid inexperienced data analysts in defining reliable simulation experiments.

Web Servers and Web Traffic In the literature, there are several papers on performance analysis of web servers. Many of these papers present measurement studies that focus on workload characterisation [6, 14, 82] or measurement of, e.g., resource utilization and response time [2, 41]. Analytic models have been used to analyse the performance of HTTP over several transport protocols [61] and for capacity planning of web servers [41]. As one of the few simulation studies of web servers, [123] presents an end-to-end queueing model of a web server environment. These studies provide excellent insight into the performance of web servers, and have made significant contributions to the understanding of web workloads. Our work provides a modelling framework that can be used for both performance analysis and functional analysis of web servers, in particular, and of distributed systems, in general.

Chapter 6

Conclusions and Future Work

This chapter concludes Part I of this dissertation. Section 6.1 summarises the main contributions of this dissertation. Section 6.2 presents a number of directions for future work.

6.1 Summary of Contributions

As stated in Sect. 1.4, the goal of my research has been to investigate and facilitate the practical use of CP-nets for performance analysis of realistic, industrial-sized systems. The research presented here contributes to the development of tools, theory, and practical application of CP-nets for performance analysis. The main contributions of the work are summarised below.

- The development of facilities supporting reliable, simulation-based performance analysis using CP-nets [126]. The facilities provide means for running independent simulations, collecting data from each simulation, saving simulation output systematically, and calculating confidence intervals for estimates of performance measures. Additional support is provided for defining, simulating, and comparing different configurations of a given CP-net.

- The development of facilities for monitoring simulations of CP-nets [90]. With monitors it becomes possible to make an explicit separation between modelling the behaviour of a system and monitoring the behaviour of the model. As a result, cleaner and more understandable CPN models can be created. Monitors provide a common interface for mechanisms that inspect and control simulations of CP-nets. Monitors can be used for diverse purposes, such as animation and visualisation, communication between CPN simulators and other processes, and data collection. The monitoring framework served as a basis for implementing several standard data collection monitors.

- A novel method for adding auxiliary information to CP-nets [89]. The main contribution of this paper is the definition of annotations which can be used to add auxiliary information to a CP-net in order to facilitate

the use of the CP-net for a particular purpose. The main advantages of using annotations are: annotations are defined separately from a CP-net, annotations affect the behaviour of a CP-net in a very limited and predictable manner, and it is easy to enable and disable one or more layers of annotations.

- Performance analysis of a web server using CP-nets [127]. This case study has shown that it is relatively easy to analyse the performance of an industrial-sized system using CP-nets and the newly developed performance facilities. An advantage of using Design/CPN is that it allowed us to use the same workload for both the actual web server and the CPN web model during the calibration phase of the project. The case study demonstrated that typical CPN users are not experienced performance analysts, and that this fact ought to be taken into consideration when developing performance-related facilities for CPN tools.

Several new facilities and tool extensions that have been developed in conjunction with the research presented in this dissertation. The new performance facilities (batch scripts, output management facilities, etc.), which build upon the Performance Tool, have been implemented as an extension to Design/CPN Version 4.0.x. These facilities are complete enough to be usable, but they are not polished enough to be included in a released version of Design/CPN. The facilities can be obtained by contacting the author. Prototype monitoring facilities have been implemented for CPN Tools. These facilities include the basic functionality for creating data collection and simulation breakpoint monitors from user-defined `check` and `observe` functions. Several standard data collection monitors have also been implemented. The prototype monitoring facilities lack the following important components: a GUI for defining monitors, facilities for reporting on syntactical errors that are found in user-defined `check` and `observe` functions, and support for MSC monitors.

6.2 Future Work

Coloured Petri nets are well suited for modelling and analysing real-world systems, and they have the potential for being used for performance analysis, but in practice they rarely are. In the past, each individual user was responsible both for incorporating data collection functionality in their models and for applying proper statistical techniques when analysing simulation output. After the introduction of the Performance Tool, users no longer need to define their own data collection functionality. I believe that the primary obstacle in using CP-nets for performance analysis is the lack of performance-related tool support. A number of ideas for future work which could help to remedy this problem are discussed below.

The most obvious first step towards alleviating the problem of insufficient tool support for performance analysis is to consolidate and integrate the new performance facilities that are discussed in Chapters 2 and 7 in one CPN tool.

Since CPN Tools is poised to become the successor to Design/CPN, these facilities must be integrated in CPN Tools. Not only should the facilities be integrated in CPN Tools, they must also be improved. For example, there is currently no explicit support for analysing the steady-state behaviour of a CP-net. It is only possible to run a given number of independent, terminating simulations. To fix this problem the batch script for simulating one configuration of a CP-net could be extended relatively easily to provide support for running non-terminating simulations. Simple techniques are also available for calculating confidence intervals to obtain a user-specified or relative precision. An effort has been made to create facilities for *standard* data collection and analysis, i.e. these facilities do not require any programming by the user. Incorporating standard data collection monitors in CPN Tools would also be beneficial. The introduction of proper types of simulation output and support for sound statistical techniques will aid inexperienced data analysts in doing performance analysis of systems.

If the interest for using CP-nets for performance analysis increases, then there are a number of ways in which the tool support for performance analysis could be improved. High-level support could be provided for sensitivity analysis, gradient analysis, or optimisation techniques. Sensitivity analysis investigates how extreme values of parameters affect performance measures, while gradient estimation examines how small changes in parameters affect the performance of the system. Optimisation is often just a sophisticated form of comparing alternative configurations, in that it is a systematic method for trying different combinations of parameters in hope of finding the combination that gives the best results.

Additional support could also be developed for comparing alternative configurations of CPN models. Chapter 2.2 discussed a batch script which can be used run a number of simulations with the purpose of comparing the performance of alternative configurations of a CP-net. A similar script was used in a recent Australian PhD dissertation [55] to specify a number of configurations of a CP-net and then to carry out state space analysis for each configuration. It would be interesting to examine the similarities between these two scripts and to investigate the usefulness of providing generalised support for comparing alternative configurations by means of different kinds of analysis.

Better support for monitors and annotations should also be provided. A number of improvements for the monitoring facilities are outlined at the end of the previous section, i.e. in Sect. 6.1. As mentioned previously, annotations have not been used in practice. It would be interesting to implement prototype support for annotations in order to investigate the practicality and usefulness of the proposed method. There are a number of techniques that are employed in the GUI for CPN Tools that are well suited for implementing annotation layers. Beaudouin et al. [17] advocate separating objects and commands relevant to a specific activity into *layers*. For example, a CP-net could be defined in one layer, while a simulation layer could be used to show the actual markings and enabled transitions of the CP-net during a simulation. It would be quite natural to implement annotation layers in such layers. Annotated tokens could be shown when simulation and annotation layers are enabled together, while regular, non-

annotated tokens could be shown if the annotation layer was disabled during a simulation. The methods that are used for defining groups [16, 37] could also prove to be useful for defining annotation layers. When defining a group, the graphical representation of a CP-net is initially dimmed, and objects are then highlighted when they are added to the group. Similarly, when defining annotations, places and arcs could be highlighted to indicate that they are annotated places and annotated arcs.

Part II

Papers

Chapter 7

Performance Tools for Coloured Petri Nets

The paper "Performance Analysis using Coloured Petri Nets" presented in this chapter has been accepted for presentation at the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02). A shorter version of the paper will appear [126].

[126] Performance analysis using coloured Petri nets. To appear in the proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02), 2002.

This chapter contains the paper that was submitted to MASCOTS'02. The paper that was submitted to the symposium is longer than the final paper [126].

Performance Analysis using Coloured Petri Nets

Lisa Wells*

Abstract

This paper provides an overview of improved facilities for performance analysis using coloured Petri nets. Coloured Petri nets are well suited for modeling and analyzing large and complex systems for several reasons: hierarchical models can be constructed, complex information can be represented in the models, it is possible to model the time used by different activities in a system, and mature and well-tested tools exist for creating, simulating, and analyzing coloured Petri net models. The paper describes steps that have been taken to make a distinction between modeling the behavior of a system from observing the behavior of a model. Performance-related facilities are discussed, including facilities for running multiple simulations, generating statistically reliable simulation output, and comparing alternative system configurations. This paper will focus on how coloured Petri nets can be used to analyze network protocols, but the facilities discussed here can be used to analyze any kind of system.

7.1 Introduction

Performance is often a central issue in the design, development and configuration of systems. Performance analysis studies are conducted to evaluate an existing system, to compare alternative system configurations, or to find an optimal configuration of a system. Many different kinds of models, including both simulation and analytical models, are used to analyze the performance of a wide variety of systems. Some tools support performance analysis of one particular kind of system, e.g. GloMoSim [8] is a simulator for wireless ad hoc networks. These tools often provide libraries of validated models from one particular domain, and the models can be combined and parameterized in order to analyze a particular system. There are also general simulation modeling packages, such as Arena [76] and SIMNET [118], that are good for analyzing the performance of many different kinds of systems, but these packages provide limited support for functional analysis of a system.

A large body of research is concerned with using formal methods for performance analysis [23], this includes research on Petri nets [108], stochastic activity nets [110] and process algebras [62]. Most of the current research concerning formal methods and performance analysis use analytical methods, such as Markov

*Department of Computer Science, University of Aarhus, Åbogade 34, 8200 Århus N, Denmark. E-mail: wells@daimi.au.dk.

processes, for performance analysis. An advantage of using analytical models is that they can provide definitive answers regarding the performance of a model. However, for even small configurations of a system it may be impossible to generate the analytical models needed for performance analysis due to the state explosion problem. Furthermore, it can be difficult to create accurate and understandable models of industrial-sized systems when using, e.g. low-level Petri nets.

Coloured Petri nets [70, 80] (CP-nets or CPN) are well suited for modeling and analyzing large and complex systems for several reasons: hierarchical models can be constructed, complex information can be represented in the token values and inscriptions of the models, it is possible to model the time used by different activities in a system, and mature and well-tested tools exist for creating, simulating, and analyzing CPN models. There are relatively few studies that focus on performance analysis using high-level Petri nets, and even fewer that focus on simulation-based performance analysis using high-level CP-nets. Simulation has primarily been used for debugging, validation, and checking of logical correctness, and as a result, there is minimal support for simulation-based performance analysis using high-level Petri nets. This paper will discuss how coloured Petri nets and simulation can be used for analyzing the performance of industrial-sized systems.

This paper provides an overview of improved facilities for performance analysis using coloured Petri nets. This paper will focus on how CP-nets can be used to analyze network protocols, but the facilities discussed here can be used to analyze any kind of system. Network protocols are particularly interesting because it is often important to analyze both the functionality and the performance of a protocol. The functionality of a protocol can be analyzed to determine whether the protocol provides the service that it should, e.g. reliable, stream-based communication over unreliable networks, and whether there are ambiguities or unforeseen problems with a protocol specification. Performance analysis of a network protocol can examine, for example, how well the protocol handles different types of traffic, how fast the protocol responds to changes in the environment, as well as packet delay and utilization of communication channels.

CP-nets and the Design/CPN tool [31, 40] have been used to analyze many computer and telecommunication systems. Verification studies have analyzed the functionality of protocols, e.g. WAP [56], RSVP [125] and BeoLink [30], and intelligent networks in telephone systems [25]. A few performance studies have also been done, e.g. in the areas of web servers [127], ATM network algorithms [36], and comparisons of TCP implementations [38].

The paper is structured as follows. Section 7.2 introduces a CPN model of a stop-and-wait protocol that will be used as an example throughout the paper. Section 7.3 discusses steps that have been taken to separate modeling the behavior of a system from analyzing the behavior of the system. Section 7.4 presents the improved facilities for Design/CPN and related tools that support data collection and performance analysis using CP-nets. Finally, Sect. 7.5 discusses related work.

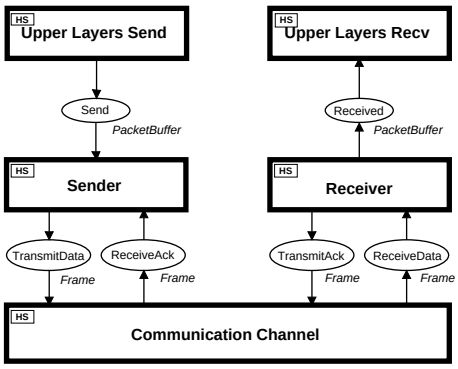


Figure 7.1: CPN model of stop-and-wait protocol.

7.2 Example: Stop-and-Wait Protocol

This section presents a CPN model that will be used as a running example throughout the rest of the paper. The example is a model of a stop-and-wait protocol from the data link control layer of the OSI network architecture. The protocol is quite simple, but it is sufficient for introducing the most interesting new concepts related to using CP-nets for performance analysis of network protocols (and other systems). A detailed description of the model is not provided here, but a thorough description of the model can be found in [80] which provides a general introduction to coloured Petri nets and related analysis methods. The following description of the model is taken from [80].

Figure 7.1 shows an overview of a timed, hierarchical CPN model of the stop-and-wait communication protocol. The system consists of a sender transmitting data packets to a receiver across an unreliable, bi-directional communication channel. The sender accepts data packets from protocols in the upper layers of the OSI network architecture. Similarly, the receiver passes packets that have been properly received to the upper layers of the protocol stack. The Upper Layers Send module generates workload for the Sender module in this CPN model. The Communication Channel is a module that provides a simple model of an unreliable network in which packet loss and overtaking can occur. The stop-and-wait protocol is modeled in detail in the Sender and Receiver parts of the model.

Figure 7.2 shows the Sender part of the CPN model. The states of a CP-net are represented by a number of *tokens* positioned on *places*, which are drawn as ellipses. Each token carries a data value, such as an integer or a string. The events of a CP-net are represented by means of *transitions*, which are drawn as rectangles. Incoming packets from the upper layers are added to a packet buffer (Send). The sender can only accept (Accept) a new packet from the upper layer after an acknowledgment has been received for the previous packet. If an acknowledgment has been received, then status of the sender (NextSend) indicates the sequence number for the next packet. When a packet is accepted, a sequence number is added to the packet to form a data frame which then is ready to be sent (Ready). The status of the sender is changed to reflect that

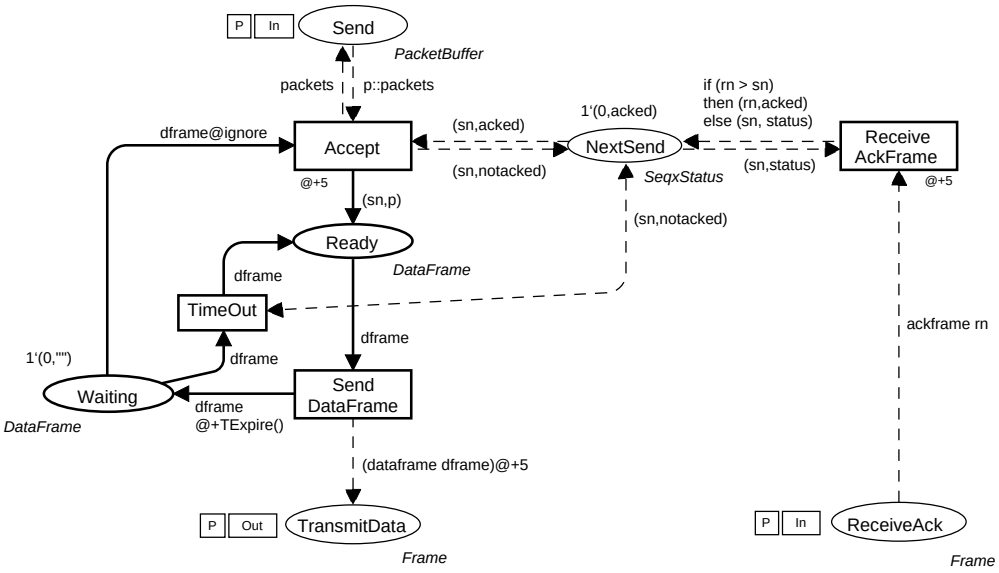


Figure 7.2: Sender module of stop-and-wait CPN model.

it is sending a packet that has not yet been acknowledged. It takes 5 units of time to process each incoming packet from the upper layers.

When a data frame is sent (Send DataFrame), two things happen. The data frame is transformed to a frame which is sent to the network interface (Transmit Data). It takes 5 units of time to complete this process. At the same time, a copy of the data frame is retained at the sender (Waiting) in case the data frame needs to be retransmitted later. The function TExpire() determines how long the sender will wait before retransmitting the data frame. If the time out period expires before an acknowledgment is received, then the data frame is retransmitted (TimeOut). In this model, unlimited retransmissions are allowed.

When the receiver receives a frame, it sends an acknowledgment back to the sender indicating the sequence number of the next packet that it expects to receive (rn). Thus, when the sender receives an acknowledgment (Receive AckFrame) it must compare the sequence number in the acknowledgment with the sequence number of the packet that it last sent (sn). If rn>sn, then the receiver has properly received the most recent packet, i.e. the packet with sequence number sn has been acknowledged, and the sender is free to accept the next packet from the upper layers. On the other hand, if rn≤sn then the status of the sender remains unchanged since an acknowledgment for an old packet has been received. Processing an acknowledgment also takes 5 units of time.

The above primarily describes the functionality of the system. Let us consider how CP-nets can be used to analyze the performance of the system. We have mentioned that the model contains information about how much time certain activities take. In the Upper Layers module, packets are generated on-the-fly during a simulation. A parameter in the model indicates whether the periods between packet arrivals are constant or exponentially or uniformly distributed. Another parameter indicates the average amount of time that passes between

the arrival of two successive packets. Similar parameters are used for determining the network delay and network reliability in the Communication Channel.

For this system, there are several performance measures of interest, including average queue length of packets waiting to be sent, average packet delay and network utilization. Packet delay is the time from which a packet is put in the Send buffer on the sender side until it is properly received by the receiver. Network utilization is the percentage of time in which the network is busy transmitting data frames or acknowledgment frames. The data channel is bi-directional, and it is possible to measure the utilization of each direction of the channel. Changing the values of parameters mentioned above will can have profound effects on the performance of the system. In Sect. 7.4 we will see how the performance of systems can be analyzed using CP-nets.

7.3 Monitoring System Behavior

A number of libraries and tool extensions have been developed for Design/CPN during the past decade. These libraries include support for message sequence charts [92], updating domain-specific graphics [106], communication between the simulator and other processes [49], and data collection [87]. These facilities can be used both to inspect and to control a simulation of a CP-net. While there are many advantages to having this extra functionality, there are a number of disadvantages as well. The most serious problem is that it is often necessary to add extra information into a CPN model in order to be able to use the facilities mentioned above. This extra information can be both extra places or transitions and auxiliary values in the data types for the tokens in the CP-net. For example, it may be necessary to add an extra place and transition to a CP-net in order to create a communication channel to another process. Introducing extra information into the model may have unexpected and undesirable effects on the behavior of the model. In this section, we will discuss new facilities with which it is possible to use the libraries mentioned above without having to modify a CP-net.

7.3.1 Monitors

Ideally in a simulation tool, there should be a clear distinction and separation between modeling the behavior of the system and monitoring the behavior of the system (model). A new framework, called the *monitoring framework* [90] has been introduced in an attempt to provide inspiration for how to obtain this separation for discrete-event system simulators. One of the most important goals of the monitoring framework is to make it possible to inspect or control a simulation without having to modify a model.

The monitoring framework has been used to implement so-called *monitors* for the new CPN simulator [96] that is available in both Design/CPN and CPN Tools [16, 37], which is a new CPN tool that will be the successor to Design/CPN. A monitor is a mechanism that can observe or monitor the states and events of a CP-net during a simulation, and that can take appropriate actions based on the observations. More specifically, a monitor is activated

```
if check (event, state)
    then act (observe (event, state), state)
```

Figure 7.3: Monitoring functions.

after each step in a simulation, and if certain conditions are met, then it will observe the current state of the model and/or the most recently occurring event, and take appropriate actions based on the observation just made. Each monitor contains three functions, `check`, `observe`, and `act`, that perform these services. The relationship between these functions is shown in Fig. 7.3.

A monitor will generally have one purpose, such as updating a log file, updating a message sequence chart, or collecting data. A number of standard monitors that can be used for any CP-net have been defined. It is also possible for a user to define monitors that are tailor-made for a specific CP-net.

7.3.2 Examples of Monitors

In this section we will consider two monitors that have been created for the stop-and-wait model. The monitor in the first example is used to stop a simulation when the sender retransmits a data frame for the third time. The second monitor is used to create a message sequence chart that illustrates the transmission and reception of frames between the sender and receiver. In Sect. 7.4 we will see how monitors can be used for analyzing the performance of the stop-and-wait protocol.

Simulation Breakpoints

Monitors can be used to control a simulation, for example, by defining breakpoints during a simulation. Any number of monitors can be defined for a given CP-net, and monitors can interact with each other. For example, one standard monitor can count the number of times the sender retransmits a data frame, a second monitor resets the first monitor each time a new packet is accepted from the upper layers, and a third monitor can stop a simulation when it observes that the sender has retransmitted a data frame three times. In this case, the third monitor has to access the counter of the first monitor in order to determine if three retransmissions have occurred.

Let us see how these three monitors are defined. It is easy to create the standard monitor that counts how many times the sender retransmits packets. In this case the user must just select, i.e. click on the appropriate node in the GUI of the CPN tool, the TimeOut transition in the CP-net and then indicate that a simple counter monitor must be created. Assume that the monitor is named `NumOfTimeOuts`. This monitor will increment a counter each time the sender retransmits a packet. This counter cannot be used alone because it will count all retransmissions of all packets, i.e. it does not differentiate between retransmissions of different packets. As mentioned above, a second monitor

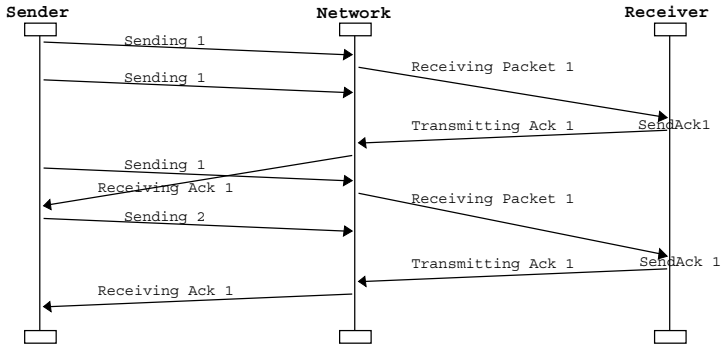


Figure 7.4: Message sequence chart.

must be created for reseting the counter monitor each time a new packet is accepted. In this case the user must select the Accept transition and then indicate that generic template code for the three monitoring functions from Fig. 7.3 must be generated. This template code must then be modified by the user in order to obtain the desired functionality. In this case, the `check` and `observe` functions do not need to be changed, and the user will only have to add `NumOfTimeOuts.reset()` to the body of the `act` function in order to reset the first monitor. An example of code for a monitor will be shown in the next section. The final monitor is defined in a similar manner, but template code can be generated for simulation breakpoints rather than generic template code. Only template code for a `check` function is generated, and the user has to add `(NumOfTimeOuts.Count())=3` to the body of the function. When these monitors are used during a simulation of the stop-and-wait CP-net, the simulation will stop if the sender retransmits a particular packet three times. It is possible to resume a simulation after it has been stopped at a breakpoint. With monitors it is very easy to define a domain-specific, simulation breakpoints without having to modify the CP-net either by adding extra places and arcs, or by adding extra information to certain events.

Message Sequence Charts

Monitors can also be used to create message sequence charts. MSCs are particularly useful when analyzing the behavior of network protocols, since they can be used to illustrate the transmission of messages. They are also very useful for debugging CPN models. Figure 7.4 shows an MSC that was generated during a simulation of the stop-and-wait model in Design/CPN. The MSC contains a new type of arrow which are called *two-step arrows* [89]. The difference between two-step arrows and the original arrows in MSCs is that the arrows are drawn differently. An ordinary arrow is drawn when one event occurs. In contrast, a two-step arrow can be drawn after two different events have occurred. The slope of the arrows indicate the passage of time; arrows in traditional MSCs are always horizontal.

When using monitors to draw MSCs, all of the functions that are used

to update an MSC are gathered together in one place. With monitors, it is possible for the user to get an overview of the places and transitions that a monitor observes without having to examine all places and transitions in the CP-net. With monitors it is also very easy to inspect the state of a CPN model during a simulation.

7.4 Performance Analysis using CP-nets

This section discusses the facilities that provide support for doing simulation based performance analysis using CP-nets. A tool developer must understand the needs of users in order to create a tool that can be used in practice. Personal experience using several different simulations tools with students with mixed backgrounds (operations research and computer science) has shown that people with different backgrounds have very different needs with regards to simulation output. Inexperienced data analysts will have a tendency to believe what a tool tells them, therefore care must be taken to avoid generating misleading simulation output. More experienced data analysts generally require that more sophisticated kinds of data are generated for specific purposes. Therefore, developers of tools supporting performance analysis need to strive to provide simulation output that is statistically reliable, and that is useful for both experienced and inexperienced data analysts. These observations have influenced the development of the performance facilities for Design/CPN and CPN Tools. New facilities, that are described below, provide both support for generating statistically reliable simulation output and running multiple simulations.

7.4.1 Data Collection

The data collection facilities are based on monitors, and each performance measure is calculated by a *data collection monitor*. The simulators and performance facilities for Design/CPN and CPN Tools are implemented in Standard ML [100] (SML). Figure 7.5 shows the code[1] for a data collection monitor that calculates the average number of packets in the sender's packet buffer. The monitor is named `PacketQueue`. It is not important to understand the details of Fig. 7.5, but the functionality of the monitor will be described in general terms. In order to calculate the average number of packets in the buffer it is sufficient to measure the number of packets in the buffer only when the number of packets changes, i.e. either when a new packet is added to the buffer or when a packet is removed from the buffer. The `Event` data type shows that this monitor only observes two kinds of events: Generate and Accept events. The monitor will be activated only when the transitions Generate (in the Upper Layers Send module, not shown) and Accept occur. For this monitor, the `check` function only needs to return `true` each time it is evaluated, i.e. it will return true when either the Generate event or the Accept event occurs. The `observe` function is then called to measure the number of packets in the buffer, and this is achieved by measuring the length of the one list that is found on the place Send. Most of the code in

[1] The code in Fig. 7.5 has been edited for clarity.

```
structure PacketQueue  =                                              1
  struct                                                              2
    datatype Event =                                                  3
        Generate of int * {i, packets}                                4
      | Accept of int * {sn, packets, p, dframe}                      5
                                                                      6
    fun check (event, (SendMark: PacketBuffer list)) =                7
          true;                                                       8
                                                                      9
    fun observe (event, (SendMark: PacketBuffer list)) =              10
        if (length SendMark)>0                                        11
          then length(hd SendMark)                                    12
        else 0;                                                       13
                                                                      14
    fun act (observedval, (SendMark: PacketBuffer list)) =            15
        DataCollection.update(index, observedval);                    16
                                                                      17
    fun monitor (event: Event, moduleInstance) =                      18
        let                                                           19
          val SendMark =                                              20
            PlaceSend.getMarking(moduleInstance)                      21
        in                                                            22
          if check(event, SendMark)                                   23
            then act(observe(event, SendMark),                        24
                     (SendMark))                                      25
          else ()                                                     26
        end;                                                          27
  end                                                                 28
```

Figure 7.5: Data collection monitor.

Fig. 7.5 is generated completely automatically. The user is only responsible for defining the bodies of the `check` and `observe` functions, represented by lines 8 and 11-13, respectively. In order to define a data collection monitor, template code can always be generated, and it is up to the user to make any necessary modifications to the code. In many cases, it will not be necessary to make any changes at all.

Standard data collection monitors that can be used to calculate a variety of different performance measures have been defined. For example, one standard data collection monitor calculates the average number of tokens on a place during a simulation. The average number of tokens on a place could represent, e.g. average queue length, utilization of processors in a multi-processor system, or average available buffer space. Counting the number of times a particular event occurs has also be implemented as a standard monitor, and this could be used for counting the number of retransmissions or the number of packets lost. Data collection monitors provide a more intuitive interface for accessing performance measures than what was previously available in Design/CPN.

The individual data values that are observed by a data collection monitor can be saved in a log file, referred to as an *observation log file*, and can be used to

calculate statistics. Three kinds of statistics can be calculated: tally statistics[2], time-persistent statistics[3], and counter statistics. Examples of tally statistics are average, minimum and maximum packet delays. The average number of packets in the packet buffer is an example of a time-persistent statistic. Counter statistics are simple sums, and the monitor `NumOfTimeOuts` from Sect. 7.3.2 is a good example of a counter statistic (assuming that it is not reset each time a new packet arrives).

7.4.2 Analysis of One System

Performances measures that are calculated via simulation modeling are generally only estimates of the true performance measures. One of the dangers of using simulation for performance analysis is accepting the output statistics from a single simulation of a model as the "true answers" [84]. To compound the problem even further, analysis of output data from one simulation is sometimes done using statistical formulas that assume independence when in fact the data is dependent, and is a problem in the Design/CPN Performance Tool [87] (also referred to as the Performance Tool). New features have been developed that provide support for properly analyzing the behavior of a system.

Multiple Simulations One of the desirable features for simulation modeling tools is a single command to make several simulation runs (replications) of a given model. No such command currently exists in Design/CPN, but the simulator contains a number of SML functions that can be used to, for example, initialize the state of the monitor, run simulation, and collect data. Several of these primitives have been combined to create a simple batch script (which is just an SML function) which can be used to run a given number of independent, terminating simulations. Data is automatically collected and saved during each simulation. Each terminating simulation can provide one estimate for each of the performance measures that have been defined for a particular model.

Confidence Intervals *Confidence intervals* can be used to indicate how precise an estimate of a performance measure is. Given a set of estimates of a performance measure, it is easy to calculate confidence intervals. However, the estimates must be independent and identically distributed (IID) in order to calculate an unbiased estimate of the variance of the estimates of the performance measure. IID estimates of performance measures can be collected from simulations by using the batch script from above and *batch data collection monitors*. Batch data collection monitors are created before running a number of simulations, updated after each simulation, and then used to calculate confidence intervals which will be saved in a *batch performance report*.

[2]Tally statistics are called untimed statistics in the Design/CPN Performance Tool, and they are also referred to as discrete-time statistics in performance-related literature.

[3]Time-persistent statistics are called timed statistics in the Performance Tool, and they are also referred to as continuous-time statistics in performance-related literature.

Simulation Output Simulation output it crucial for performance analysis. It is used both for analyzing the performance of the system and for presenting the results of the analysis. Therefore, it is important that a simulation modeling tool generates output that is useful for data analysts. The output should also be in formats that can be used immediately because it is much better to spend time analyzing the data rather than post-processing it or converting it to a format that can be used in reports or presentations.

Several different forms of simulation output can be automatically generated. At the end of a single simulation, all statistics from the simulation can be saved in a *simulation performance report*. The simulation performance reports that are generated by the Performance Tool can contain misleading information. These reports contained the variance and standard deviation for data values that were collected from a single simulation. In most cases these values are not likely to be IID, in which case the calculated standard deviation and variances were biased estimates of the true standard deviation and variance. In the new facilities, these values are not calculated for data values that are collected during a single, terminating simulation. Both simulation and batch performance reports can now be saved in plain text, LaTeX and HTML formats, thus sparing the user from having to manually convert plain text files to either of these formats.

Additional facilities can be used to create a simple, yet organized system for simulation output. When running the simple batch script from above, all simulation output will be saved in a directory named `batch_n`, where `n` is an integer generated by the output management facilities. Figure 7.6 shows an example of the directory structure and files that are created when running a batch of three simulations of the stop-and-wait model. The directory `batch_n`

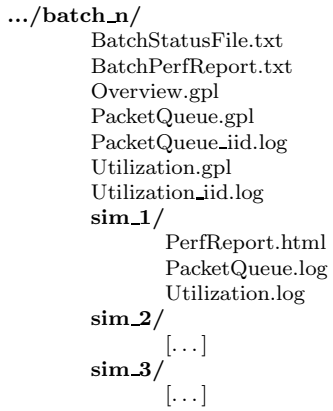


Figure 7.6: Directory structure with simulation output management.

will contain a batch status file that provides information about the status of each individual simulation. The directory also contains a group of directories `sim_1`, `sim_2`, ..., `sim_m`, where `m` is the number of simulations run in the batch. The observation files for the `i`'th simulation are saved in the `sim_i` directory, and a performance report (in the desired format) is saved here as well. After

all simulations have been run, confidence intervals can be calculated and saved in a batch performance report in the `batch_n` directory.

In addition to the systematic creation of directories for simulation output, two new kinds of files can be generated. The first is gnuplot [53] scripts. One kind of gnuplot script can be used to plot the contents of observation files in the `sim_1` to `sim_m` directories. If an observation log file named `PacketQueue.log` has been generated for each simulation, then the gnuplot script `PacketQueue.gpl` can be used to plot the observation log files named `PacketQueue.log` in the `sim_` directories. A similar gnuplot script will be generated for each other set of similarly named observation log files. The `Overview.gpl` gnuplot script will load and plot all of the other gnuplot scripts one after the other.

The other new kind of files contains data that was used for calculating confidence intervals. At the end of a simulation the `Avrg` statistic is accessed from the `PacketQueue` data collection monitor. The average is then used to update both a batch data collection monitor (with the same name) and a file named `PacketQueue_iid.log` in the batch directory. When the confidence intervals are calculated there will be an entry in the batch performance report which contains the average and 95% confidence interval for the values found in `PacketQueue_iid.log`.

7.4.3 Comparing Alternative Configurations

Simulation studies can be made for many different reasons. The purpose of some studies may be to compare the performance of several given configurations or to choose the best of the configurations. If the scenarios are not predetermined, then the purpose of the simulation study may be to locate the parameters that have the most impact on a particular performance measure or to locate important parameters in the system. *Sensitivity analysis* investigates how extreme values of parameters affect performance measures [77]. *Gradient estimation*, on the other hand, is used to examine how small changes in the parameters affect the performance of the system. *Optimization* is often just a sophisticated form of comparing alternative configurations, in that it is a systematic method for trying different combinations of parameters in hope of finding the combination that gives the best results. Inherent in all of these activities is the need to be able to run simulations for different configurations regardless of whether the configurations are very different from each other or whether there is only a slight change from one configuration to another. Comparing configurations is, in turn, dependent on running many simulations.

Another batch script has been developed for running simulations for a number of different system configurations. This batch script can be used if a new configuration can be specified by changing numerical parameters in a CP-net. The user must specify a range of values that one or more parameters should take on, and the batch script will ensure that system parameters are changed between simulations, and that a given number of simulations are run for all given combinations of parameter values. A *configuration status file* contains the values of the parameters for each configuration, and it indicates which batch directory contains the output for each configuration.

One well-known technique for comparing two alternative system configurations is to calculate the so-called *paired-t confidence interval* for the expected difference for a given performance measure. With this technique, n IID estimates of the performance measure are needed from each of the two configurations. These estimates can be obtained by running n independent replications of each of the simulations. Then the estimates from each configuration are paired, and the difference is calculated for each pair. Whether or not the performance of two system configurations are significantly different can be tested by calculating a confidence interval for the expected value of the difference between the estimates. If the confidence interval for the expected difference between performance measures contains zero, then one must conclude the two configurations are *not* significantly different, based on the available observations.

The batch script that was introduced in this section can only be used to run simulations for the different configurations. There is no integrated support for actually comparing the results of the simulations. However, the output that is generated is extremely useful for comparing two configurations, as will be shown in the next section.

7.4.4 Variance Reduction

One of the drawbacks of simulation analysis is that it can take a long time to run a simulation. This problem is amplified if many simulations need to be run in order to achieve desired confidence intervals. *Variance-reduction techniques* (VRT) can sometimes be used to reduce the number or length of simulations that need to be run. The goal of variance reduction is to reduce the variability of estimates of performance measures without affecting the expected value.

If two different system configurations are compared using different random numbers for every simulation, then it may be difficult to determine whether differences in performance measures should be attributed to the use of different random numbers or to actual differences in the system configurations. Using *common random numbers* (CRN) when comparing alternative system configurations is a useful and practical variance-reduction technique. The idea of CRN is to use the same source of randomness, i.e. the same random numbers, for each of the configurations being studied. The effects of CRN may be improved if the simulator can be forced to use the same random numbers for the same purpose in each configuration. This process is called *synchronizing* the random numbers. Not only is CRN a useful statistical technique because it may mean that fewer or shorter simulations can be run in order to achieve the desired precision of estimates, but it also implies that a fairer comparison of the configurations can be made because the experimental conditions are the same for each configuration [54]. Support has been added for using CRN when simulating CP-nets precisely because of the appeal of this notion of fairer comparisons.

The easiest way to implement synchronization is to use different sources of random numbers for each random input process. Therefore, the random number generator that is used to generate random variates in Design/CPN has been modified, such that it can provide 10 streams of random numbers with one million independent random numbers in each stream. The random seeds

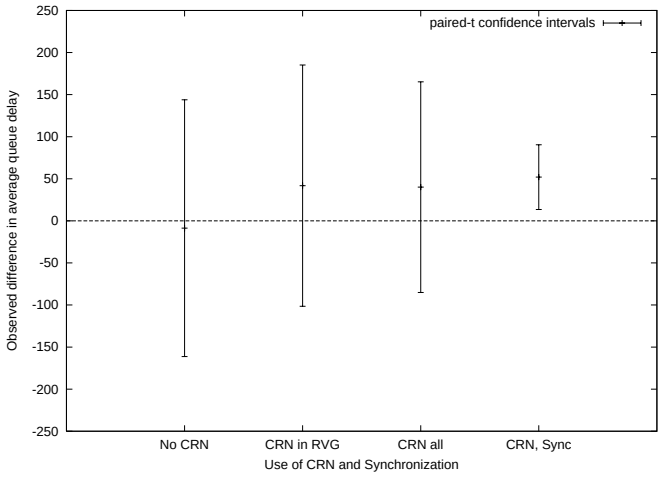


Figure 7.7: Effects of CRN when comparing system behavior.

for each of the streams can be reset. By using a separate stream for each source of randomness in a model, it is possible to achieve a certain degree of synchronization for simulations of two different system configurations.

In Design/CPN and CPN Tools there are two sources of random numbers. One random number generator is used to select which event should occur when there is more than one event that can happen at a given time. The second random number generator is used to implement the random variate generators. This means that the second random number generator is used to generate both interarrival times and network delays in the stop-and-wait model. By using one stream for generating arrival times and another stream for network delays, it is possible to achieve a certain degree of synchronization for simulations of two different configurations of the stop-and-wait model.

The effects of CRN will be illustrated by comparing two different configurations of a CPN model[4] of a queuing system. Comparisons of the two configurations were made in which varying degrees of CRN and synchronization were used. Analytical methods can be used to show there is a significant difference in the performance measure for the two given configurations [84]. In this example, we will see whether we can draw the same conclusion based on simulation output from 20 simulations of each configuration, and we will see how the outcome of the comparison can be affected by using various degrees of CRN.

Figure 7.7 shows the paired-t confidence intervals that were used to compare the configurations. In the first three comparisons, there was no attempt to synchronize the use of random numbers. In the comparison labeled No CRN, common random numbers were not used at all, and the confidence interval contains zero. Therefore, based on the available observations, one must conclude that the two system configurations are not significantly different. For the com-

[4]The example is taken from Chapter 11 in [84] and compares the average queue delay for the first 100 jobs in an M/M/1 queuing system and an M/M/2 queuing system.

parison labeled CRN in RVG, common random numbers were used for generating random variates, but independent random numbers were used for selecting the order in which concurrently enabled events should happen. A comparison of these two configurations also reveals that they are not significantly different, but the confidence interval for the difference in their performances is slightly shorter than that of the No CRN comparison. This indicates that there was a slight reduction in variance when some common random numbers were used. Using CRN both for selecting events and for generating random variates leads to a similar reduction in length of the confidence interval, as can be seen for the comparison labeled CRN all. However, the conclusion of the comparison must still be that the two systems are not significantly different. Both CRN and synchronization of random numbers were used for in the comparison labeled CRN, Sync. This is the only comparison from which one can properly conclude that the performance of the two system configurations is significantly different based on the available observations. Furthermore, it is also possible to determine which configuration is better based on the average difference between the two configurations.

The paired-t confidence intervals were calculated using the files from the batch directories that contain IID estimates of performance measures from each simulation. The paired-t confidence intervals were calculated by post-processing these files with an external application. The post-processing of data took less than 15 minutes, and this is due to the fact that the necessary data was readily available and easy to import into an external program.

7.5 Conclusion and Related Work

This paper has presented an overview of improved facilities supporting simulation-based performance analysis using coloured Petri nets. With monitors it is possible to make an explicit separation between modeling the behavior of a system and observing the behavior of a system. As a result, cleaner, more understandable CPN models can be created, and the risk of introducing undesirable behavior into a model is reduced. Facilities exist for running multiple simulations, generating statistically reliable simulation output, comparing alternative system configurations, and reducing variance when comparing configurations. Most of the facilities presented here have been implemented, however, some have been implemented for Design/CPN and others for CPN Tools. Therefore, not all of them work together. Since CPN Tools will be the successor to Design/CPN, a current project is working on updating and porting the facilities from Design/CPN to CPN Tools, and the performance-related facilities will be incorporated into CPN Tools as part of this project.

There are many other tools that support performance analysis using different types of Petri nets [104]. GreatSPN [1, 29, 57] supports both low-level Petri nets and stochastic well-formed nets, which comprise a subset of CP-nets. It uses sophisticated analytic models to calculate performance measures, and simulation-based performance analysis is also an option. The performance measures that can be calculated are model-independent, e.g. it is possible to

calculate the average number of tokens on places, the probability that a token will contain a given number of tokens, and the average throughput of tokens. No support is provided for comparing alternative system configurations, and few facilities are available for visualizing the behavior of a model.

UltraSAN [120, 113] and its successor Möbius [33] support the use of both simulation and analytic methods for performance analysis using stochastic activity networks (SANs). Studies can be defined for comparing alternative system configurations, and simulation output is saved systematically in groups of related directories and files. SANs are similar to low-level Petri nets, which means that it can be difficult to create, debug, and validate SAN models of industrial-sized systems.

Ex*Spect* [122] is a CPN tool that is, in some respects, similar to Design/CPN. In contrast to Design/CPN, a number of libraries of frequently used modules is provided with the tool. It is relatively easy to build a CP-net using these modules. With Ex*Spect* it is also possible to calculate model-dependent performance measures by examining token values, and MSCs can also be generated. However, all information that is used for calculating performance measures and updating MSCs must be hard-coded directly in a model, and there is no support for running multiple simulations.

A general-purpose simulation tool such as Arena [76] provides sophisticated and excellent support for analyzing the performance of many kinds of systems. With such a tool it is possible to analyze the behavior of systems using both terminating and non-terminating simulations, to compare alternative system configurations, and search for optimal system configurations. However, it is virtually impossible to analyze the functionality of a system using such a simulation package.

There are certain disadvantages associated with using simulation based performance analysis: no definitive answers can be provided, and it may take a long time to run enough simulations in order to calculate sufficiently accurate performance measures. However, it is the best alternative for analyzing the behavior of industrial-sized models.

Chapter 8

Monitoring Simulations

The paper "Towards a Monitoring Framework for Discrete-Event System Simulations" presented in this chapter has been accepted for presentation at the 6th International Workshop on Discrete-Event Systems 2002 (WODES'02) [90].

[90] B. Lindstrøm and L. Wells. Towards a monitoring framework for discrete event-system simulations. To appear in the proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02), 2002.

This chapter is, except for minor typographical changes, the same as the paper [90].

Towards a Monitoring Framework for Discrete-Event System Simulations

Bo Lindstrøm* Lisa Wells*

Abstract

This paper presents a framework for tools for monitoring discrete-event system models. Monitoring is any activity related to observing, inspecting, controlling or modifying a simulation of the model. We identify general patterns in how ad hoc monitoring is done, and generalise these patterns to a uniform and flexible framework. A coloured Petri net model and simulator are used to illustrate how the framework can be used to create various types of monitoring tools. The framework is presented in general terms that are not specific to any particular formalism. The framework can serve as a reference for implementing different types of monitors in discrete-event system simulators.

8.1 Introduction

A variety of formalisms, e.g. finite-state machines [64], statecharts [59], and Petri nets [108], exist and are used in practice for modelling and analysing discrete-event systems. Furthermore, mature and well-tested tools exist for building and analysing models based on these formalisms. Such tools are primarily focused on providing support for the formalism and related analysis methods, such as simulation or state space exploration. However, in many situations it has proven to be useful to be able to augment rigorously based tools with additional functionality that is not directly related to the formalism. For example, during a simulation of a high-level Petri net model it can often be useful to examine the states and events of the system, periodically extract information from the states and events, and then use the information for very diverse purposes, such as: stopping the simulation when a certain state is reached, visualisation of behaviour using message sequence charts [67] (MSC), or data collection for performance analysis.

Based on our experiences with implementing and using Design/CPN [40] which is a tool for coloured Petri nets (CP-nets or CPN) [70, 71], we have observed that the design and implementation of efficient and effective tool support for a specific formalism is generally focused on the formalism, while extracting information for other purposes is typically done using ad hoc methods. That

*Department of Computer Science, University of Aarhus, Åbogade 34, 8200 Århus N, Denmark. E-mail: blind,wells@daimi.au.dk.

means that for each different kind of information that can be extracted from a simulation and processed, a new mechanism is implemented for extracting the information. Some of these ad hoc methods are directly reflected in the models, e.g. it becomes necessary to add new events that are used solely to extract information. This can introduce errors into the models and is undesirable.

Even though the extracted information may be used for different purposes, the way the information is extracted is often similar. This means that it is possible to create a general mechanism for defining how to extract information from a model. In this paper, we will use the term *monitor* to denote any mechanism which inspects or monitors the states and events of a discrete-event system model, and which can take an appropriate action based on the observations. For example, a monitor of a communication protocol model could inspect the events during a simulation of the model and update a message sequence chart each time an event corresponding to the transmission of a message takes place.

The purpose of this paper is to present a general monitoring framework for discrete-event system simulators that can be used to standardise monitors within a given tool and to unify interaction with monitoring facilities. In other words, we present a flexible framework that can be used for defining many different types of monitors. It is our experiences with implementing the data collection facilities [87] and using other ad hoc monitoring techniques in Design/CPN that has inspired us to create the monitoring framework. The data collection facilities were designed and implemented such that they could be used without having to make any modifications to a model. One of the goals of the monitoring framework is to make it possible to use monitors to inspect or control a simulation without having to alter models. With monitors it becomes possible to make an explicit separation between modelling the behaviour of the system and monitoring the behaviour of the model.

There are several advantages of using a common framework for defining monitors. One advantage of having a common interaction technique for all monitors in one simulator is that it may be easier for users to learn and use a variety of existing monitors. We also believe that the use of standards improves the extensibility of tools. In other words, it should become easier to add new monitoring techniques without using ad hoc solutions, and the implementation of new monitors may be simpler due to reuse of code.

Flexible and standardised monitoring facilities should also make it easier to extend the use of monitoring to a wider area, by making it easier to define and integrate new monitors into a tool using the monitoring framework. In addition, we believe that a standardised and common approach where the monitoring, to some extent, is independent of the model itself will extend the usability of analysis tools for discrete-event systems. For example, using monitors for communicating with external processes or for updating domain-specific graphics may extend the use-domain of formal methods, as it becomes possible for people unfamiliar with a given formalism to use monitors to interact with a "black box" containing the formalism in order to do system analysis.

The framework will be described using general terms from discrete-event systems. When we discuss concrete monitors, coloured Petri nets will be used as a representative example of a formalism for modelling and analysing discrete-

event systems. It should, however, be easy to translate the meaning to any discrete-event system formalism with concepts for states and events.

The paper is structured as follows: Section 8.2 motivates the framework by presenting a CPN model of a communication protocol. The model is used to illustrate some of the monitors which can be used, e.g. to gain knowledge about the behaviour of the model. Section 8.3 presents the general monitoring framework for simulation models of discrete-event systems. Section 8.4 describes how the monitors presented in Sect. 8.2 can be realised using the general framework presented in Sect. 8.3. Finally, Sect. 8.5 concludes and gives directions for future work.

8.2 Example: Monitoring a Communication Protocol

In this section, we will present an example model and then use it to illustrate how different monitoring tools can be used. Even though the model itself is fairly simple, many realistic and practical monitors can be used to inspect and modify the state of the model during simulations. The model is taken from [71] which contains a detailed description of both the model and timed coloured Petri nets.

8.2.1 The Communication Protocol

The example that we will consider is a model of a simple communication protocol. This protocol is used to ensure reliable transmission of packets across an unreliable network, i.e. a network in which overtaking and packet loss can occur. Each packet includes a sequence number which is used to ensure that packets are received once and only once, and in the proper order. After a packet has been received, an acknowledgement is returned to the sender. This acknowledgement contains the sequence number of the packet that the receiver expects to receive next. Since both packets and acknowledgements can be lost, the sender uses a timeout mechanism to trigger periodic retransmission of a packet which has not yet been acknowledged.

Figure 8.1 shows a timed CPN model of this communication protocol. The model was created in Design/CPN, in which prototype monitoring facilities have been developed. The model consists of a *Sender* part (left-hand side), the *Network* part (middle), and a *Receiver* part (right-hand side).

The sender sends packets (Send Packet), where a packet consists of a sequence number and the data to be sent. Moreover, the sender has a counter (NextSend) which indicates the number of the next packet to be sent. This counter is updated whenever an acknowledgement is received (Receive Ack). A constant timeout period of wait units of time is defined, and a packet is retransmitted if an acknowledgement is not received within this period of time.

The network transmits packets (Transmit Packet) from the sender/network interface (A) to the network/receiver interface (B). Similarly, acknowledgements are transmitted (Transmit Ack) from the receiver to the sender. Packet loss is

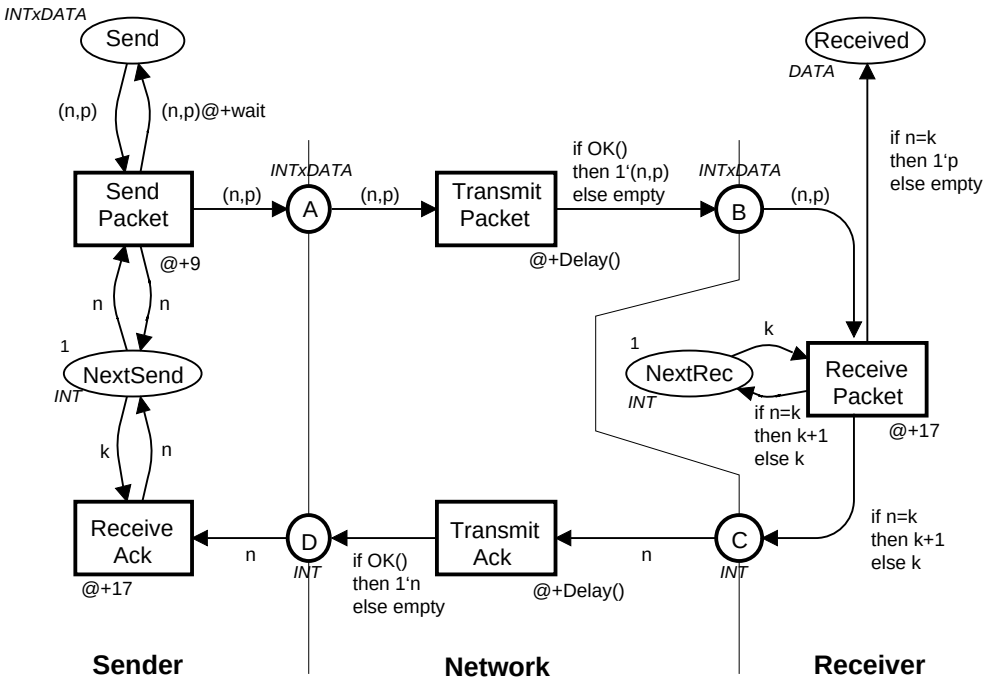


Figure 8.1: Timed communication protocol.

modelled by the OK function which is non-deterministic. The time that it takes to transmit packets and acknowledgements is determined by the Delay function.

Each time the receiver receives a packet (Receive Packet), it must decide whether to accept or reject the packet. For this purpose the receiver maintains a counter (NextRec) indicating which packet should be accepted next. If the expected packet arrives, then the packet is saved (Received), the counter is incremented, and an acknowledgement with the next expected sequence number is sent back to the sender. When an unexpected packet arrives, the packet is discarded, the counter remains unchanged, and an acknowledgement is sent back to the sender.

8.2.2 The Monitors

Several different types of monitors can be used both to control and modify a simulation of this model and to examine the behaviour of the model. All of the described monitoring activities could be incorporated directly into the model at the cost of modifying the model to contain states or events that are not found in the communication protocol.

In our work, we have observed a number of monitoring activities that can be described by the following categories, and specific examples of each category will be described in detail below. *File access monitors* are used to read and write information in files. *Simulation control monitors* are used e.g., to start and stop simulations, to determine the length of sub-simulations, or to select the order in which certain events occur. *Visualisation monitors* can be used to visualise the

```
Packet 1 received at time 190, EXPECTED
Packet 1 received at time 255, DISCARD - expecting 2
Packet 2 received at time 337, EXPECTED
```

Figure 8.2: Excerpt from the log file for the receiver.

behaviour of the system during a simulation, e.g. by updating message sequence charts or domain-specific graphics. *Performance monitors* measure and report on the performance of the system. Communication between a simulator and an external process can be controlled via *communication monitors*. Finally, *property monitors* can be used to do functional analysis of a model. Functional analysis is concerned with proving that the system behaves as expected or that certain state and/or event properties hold for the system. Let us now consider specific examples from these categories of monitors.

Simulation Stop Monitor Suppose the simulation of the communication model should be stopped after a given number of packets have been received in the correct order. This type of stop criteria is completely dependent on the model and the system, in contrast to more general and model-independent types of stop criteria such as stopping after a given number of events have taken place or after a certain amount of model time has passed. Stopping a simulation after a certain number of packets have arrived using only general stop criteria would generally require modifying the model, e.g. by adding an event that could occur only after the required packets have arrived. In contrast, a simulation control monitor could be used to inspect either the states or the events of the system and then stop the simulation when the required packets had arrived without having to modify the model.

Log-File Monitor It may be useful to maintain a receiver log file containing information about the packets that were received. For example, the log file could contain the sequence numbers of the packets that were received, the time at which they were received, as well as noting which packets were received in the correct order. Figure 8.2 shows an excerpt from a file that was generated by a log-file monitor when simulating the protocol model. Such a log file could be used for debugging purposes or for analysing the system, e.g. for analysing arrival rates. A log-file monitor is a file access monitor which can only write strings in a file.

Message Sequence Chart Monitor MSCs have proven to be useful for visualising the behaviour of communicating processes. In the context of UML, MSCs are used for specifying communication patterns, while they are often used for analysing communication patterns within the context of CP-nets. MSCs provide a high-level view of communication patterns that may not be obvious if one were restricted to inspecting every simulation event involving the communicating processes.

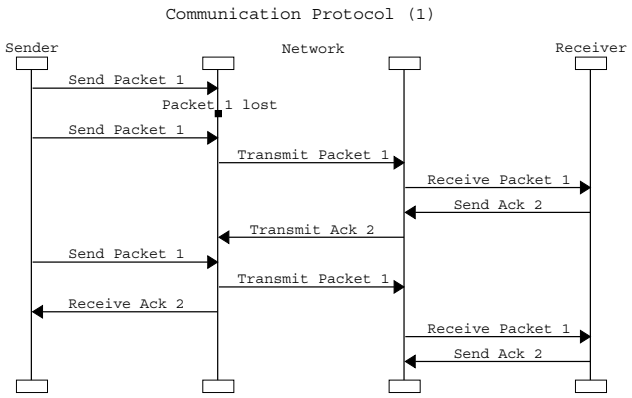


Figure 8.3: MSC for communication protocol.

Figure 8.3 is a MSC that was generated by a visualisation monitor. The MSC shows that the packet with sequence number 1 (Packet 1) was sent to the network by the sender three times. The first time Packet 1 was sent, it was lost on the network. Then Packet 1 was sent, transmitted across the network, and received by the receiver. The receiver responded by sending an acknowledgement with the sequence number for the next expected packet (Ack 2) back to the sender across the network. Finally, a timeout occurred, and Packet 1 was sent one last time before the acknowledgement from the receiver was received by the sender.

Data Collection Monitor Monitors can also be used to measure the performance of a system by collecting numerical data during a simulation. The data that is collected can be used to calculate statistics or saved in files which can be post-processed after a simulation finishes. For example, the protocol model could be used to measure the proportion of received packets that are discarded. This performance measure could be of particular interest when using simulation to find a reasonable value for the timeout period when the approximate network delay and rate of packet loss are known. A data collection monitor, which is one type of performance monitor, measuring the ratio of discarded packets among all received packets would simply have to observe the events corresponding to the reception of packets (Receive Packet), and calculate the proportion of discarded packets in an appropriate manner.

Communication Monitors Communication with external processes can be very useful when simulating models of discrete-event systems, as the external processes can augment the functionality of the simulation tool in a variety of ways. For example, an external process could provide input or workload for the model, or the external process could also be used to process data that has been extracted from the model during simulation.

Communication channels can also be used for two-way communication between a simulator and an external process. The external process could be, for

example, a real system that is built on top of the communication protocol. Alternatively, the external process could be another simulator that is simulating a model of a system that is built on top of the communication protocol.

State Property Monitor Ideally, functional analysis of a discrete-event system should be done by examining all of the reachable states and state transitions of the system, i.e. the full state space of the system. However, in many cases, it may not be possible to analyse (or even generate) a full state space due to the state explosion problem. In such a situation, an alternative, albeit somewhat unsatisfactory, solution is to determine whether or not the property holds for selected simulations using property monitors.

A state property for the communication protocol that ought to hold for all states is that the value of the counter in the sender (NextSend) should always be either less than or equal to the value of the counter in the receiver (NextRec). This property can be checked during a simulation by a property monitor that examines and compares the value of these two counters in every state during a simulation. At the end of the simulation, the monitor will report whether or not the property held.

It is important to remember that a property monitor that is used during a simulation cannot necessarily prove that a property holds for a model, as the monitor only examines one sequence of states and state transitions that correspond to a *partial* state space. While a property monitor cannot prove that a property holds, they can be helpful when debugging and validating models.

8.3 Monitoring Framework

In this section we describe a general monitoring framework which can be used to create different types of monitors in modelling tools for discrete-event systems. In Sect. 8.3.1 we give a description of the essential parts of a monitor. Section 8.3.2 presents the interface of a monitor. To use the framework in a concrete tool, the tool must be able to inspect or access both events and states of a model during a simulation.

8.3.1 Functionality of Monitors

In this section we present our proposal for the architecture of a monitor. We describe the components of a monitor and how the components interact with each other.

Architecture of Monitors In Sect. 8.2 several different monitors were presented, e.g. monitors for collecting data, maintaining message sequence charts, and communicating with external processes. By considering these monitors in detail, we identify a common pattern in how these monitors operate. Each monitor can be divided into three logical parts:

1. *Check:* When should the monitor make an observation.

```
if check (event, state)
    then act (observe (event, state), state)
```

Figure 8.4: Relation between monitor functions.

2. *Observe:* How is the observation computed.

3. *Act:* Based on the observation, what action is to be made.

As an example, consider the log-file monitor in Sect. 8.2.2. For the log-file monitor, the check determines that the file should be updated when a packet is received (Receive Packet). The observation defines the string that is to be added to the file. The action updates the file with the observed string. The state of the model is left unchanged.

Figure 8.4 illustrates the general functionality of a monitor. When, what and how an action is to be made is defined by three functions: the `check` function, the `observe` function, and the `act` function. For most monitors the following control flow takes place. First, the `check` function is evaluated to determine when an action is to be performed. When the `check` function evaluates to true, then the `observe` function is evaluated to observe a value. Finally, the `act` function uses the observed value to do a specific and monitor-dependent action. In other words, the observed value can be used to update the state of the model, the state of the simulator, and/or the environment of the simulator (e.g. file system or communication channels).

A monitor must be activated periodically during a simulation, i.e. the statement from Fig. 8.4 needs to be evaluated occasionally for each monitor. However, a monitor must only be activated at well-defined points during a simulation to avoid inconsistency. Some obvious examples of points for activating a monitor during a simulation are the following: after each event during a simulation; after certain events occurs; after a certain amount of time; after a specific number of events. Furthermore, it must be possible to examine one or more states of the model, each time the monitor is activated. In our experience with monitors for CP-nets, it has been sufficient to activate monitors *after* events have completed, at which point the current state of the model can be inspected. In other words, after an event occurs, the statement from Fig. 8.4 is evaluated for each monitor, and both the `check` and `observe` functions are able to inspect both the current state of the model and the most recently occurring event. However, the future may show that some monitors may need to be able to inspect, e.g. all previous states and occurring events, or all potential next events which can occur from the current state.

Domain and Range of Monitors We have indicated in Fig. 8.4 that a monitor must at least be able to inspect the current state of the model and the most recently occurring event during a simulation. In addition to the current state and the most recently occurring event, it is useful that both the `check`,

`observe`, and `act` functions have access to the simulator of the model, and to the simulator environment. Access to the simulator makes it possible to, e.g. stop a simulation, while access to the simulator environment may give additional possibilities such as accessing global variables, functions, and external files. Accessing the simulator environment is necessary when implementing a log-file monitor, since a log-file monitor needs to access the file system to be able to save files.

Monitors where the `act` function updates the simulator environment are only inspecting the behaviour of the system, and do not change the state of the model or the simulator. However, monitors where the `act` function modifies the state of the model or controls the simulator have to be handled with care. If it is possible to change the state of the model during a simulation, the semantics of the formalism can be violated if used improperly.

Initialising and Concluding Monitors Before a monitor can be used it may need to be initialised. In addition, after having used the monitor for monitoring a model, it may need to wrap up the work. These activities are different than the normal invocation of the monitor; they are related to initialising and concluding a monitor. For example, before the log-file monitor can write to a file, the file needs to be opened, and the file should be closed at the end of a simulation.

For several different kinds of monitors it is useful to be able to perform special activities before the `check` function is evaluated for the first time, and after the `act` function has been evaluated for the last time. In our experience the following different points of automatically initialising a monitor are useful: never, when the state of the model is initialised, or before each (sub-)simulation. Similarly, we have observed that there are two useful options for automatically concluding a monitor: never or after each simulation.

8.3.2 Interface of Monitors

In this section we provide a detailed specification for the interface of a monitor. Figure 8.5 shows the interface of a monitor, which is specified as a Standard ML [100] signature.

A monitor contains four local types: `event`, `state`, `observeType`, and `actState`. The type `event` defines what events the monitor can inspect. The `state` type defines the part of the state of the model which the monitor can inspect. The `observeType` type defines the return type for the `observe` function. The `actState` type defines the return type for the `act` function. The `actState` type is `unit` for monitors which are only inspecting the model and its environment, but which do not update the state of the model or the simulator. For monitors which can modify the state of the model or simulator, the type will be given by the type of the state of the simulator.

Once the local types of the monitor have been defined, the functions `check`, `observe`, and `act` can be created. Based on the current state and the most recently occurring event, the `check` function must be defined to return true only when the `observe` and `act` functions should be invoked. In other words,

```
datatype monitorInitMode =  NeverInit | AtInitState | AtSimStart
datatype monitorConcludeMode =  NeverConclude | AfterSim

signature MONITOR = sig
    type event
    type state
    type actState
    type observeType

    val check : event * state -> bool
    val observe : event * state -> observeType
    val act : observeType * state -> actState
    val monitor : event * state -> unit

    val init : state -> state
    val conclude : state -> state

    val init_mode : monitorInitMode
    val conclude_mode : monitorConcludeMode
end
```

Figure 8.5: Interface of a monitor.

the `check` function defines when the monitor should be activated. When the `check` function evaluates to true, the `observe` function is invoked. The purpose of the `observe` function is to inspect the state of the model or the most recently occurring event (or the environment) and to extract the value to be used by the `act` function which takes the appropriate action for the monitor.

The `monitor` function is called by the simulation tool during a simulation to activate the monitor. The `monitor` function takes care of invoking the `check`, `observe`, and `act` functions in the correct order as illustrated in Fig. 8.4. In other words, it is the responsibility of the simulation tool to call the `monitor` function often enough so that the `check` function is is able to detect the points at which the `observe` and `act` functions are to be called.

The functions taking care of initialising and concluding a monitor are specified by the `init` and `conclude` functions. As mentioned in the previous section, a monitor may be initialised and concluded at different points during its use. That information is stored in the variables `init_mode`, and `conclude_mode` using the corresponding data types `monitorInitMode` and `monitorConcludeMode`.

8.4 Concrete Monitors

In this section we will discuss how monitors for simulations of CPN models can be created based on the framework presented in Sect. 8.3. We will also discuss the advantages of providing support for both standard and user-defined monitors.

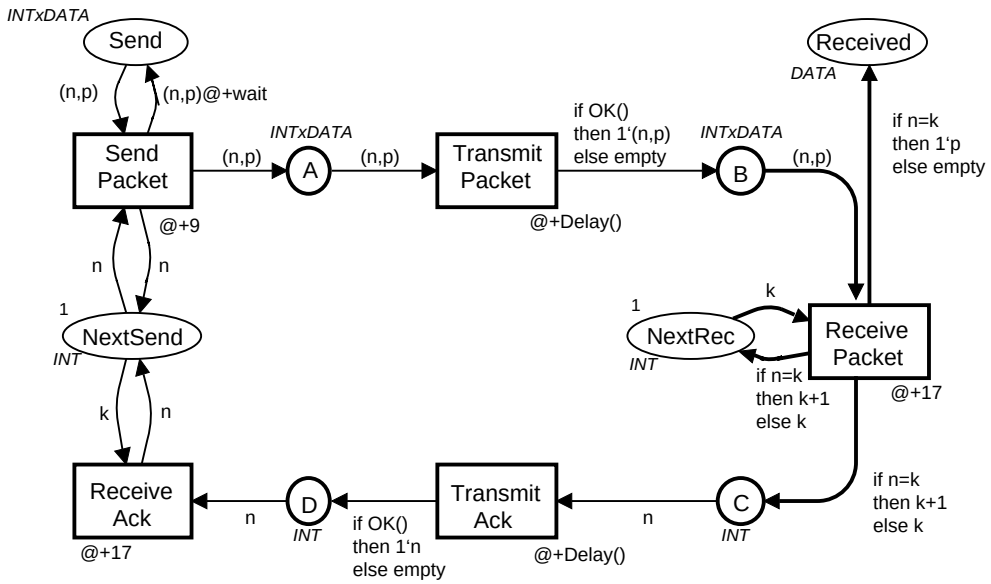


Figure 8.6: Determining events and states to monitor.

8.4.1 Creating a Log-File Monitor

In this section we describe a log-file monitor which is a monitor that can be used for any CPN model. Most of a log-file monitor can be predefined, i.e. it will be the same for all log-file monitors, independent from the model being monitored. Only minor parts of the monitor have to depend on the specific model. A log-file monitor contains predefined functions for opening, updating, and closing a file. In other words, the `init`, `act`, and `conclude` functions will be predefined. Only the `check` and `observe` functions have to be created by the user for defining when to observe the model and how to create the string to be saved.

Let us see how the log-file monitor from Sect. 8.2.2 can be created. Recall that the log-file monitor should update a file each time the receiver receives a packet. Each update should contain information that is specific for the particular packet that was received, i.e. the sequence number of the packet, the time the packet was received, and whether or not the packet contained the expected sequence number. The black parts of Fig. 8.6 indicate the parts of the protocol model which provide relevant information for creating a log-file monitor that will generate such a log file for the receiver.

In a CPN tool, it is possible for a user to select parts of a model and then have the tool automatically deduce which elements represent (part of) the state of the system and which elements represent events in the system. This is due to the fact that CPN models consist of only two types of nodes: *places* (represented by ellipses) which model the state of the system, and *transitions* (represented by rectangles) which model the events of the system. Using the information selected by a user, it is possible for the tool to generate much of the information that is required for creating a monitor that satisfies the interface

shown in Fig. 8.5. This particular log-file monitor updates a file after each event that corresponds to the receiver receiving a packet, and these events are modelled by the transition Receive Packet and the arcs surrounding the transition. The variables in the arc inscriptions contain information specific to the event of receiving a packet: `k` is the value of the counter in the receiver, and `(n,p)` is the packet that has been received, where `n` is the sequence number, and `p` is the payload or data of the packet.

```
datatype ReceiverLogEvent =                                          1
   Receive_Packet of {n: INT, p: DATA, k: INT}                       2
                                                                     3
structure ReceiverLog : MONITOR = struct                             4
                                                                     5
   type event = ReceiverLogEvent                                     6
   type state = unit                                                 7
   type actState = unit                                              8
   type observeType = string                                         9
                                                                     10
   fun check (Receive_Packet {n,p,k}, currentState) = true           11
                                                                     12
   fun observe (Receive_Packet {n,p,k}, currentState) =              13
      "Packet "^Int.toString(n)^                                     14
      " received at time"^                                           15
      timeToString(time())^", "^                                     16
      (if n=k then "EXPECTED\n"                                      17
       else ("DISCARD - expecting "^                                 18
             Int.toString(k)^"\n"))                                  19
                                                                     20
   val fileID = ref TextIO.stdOut                                    21
                                                                     22
   fun act (observeStr, currentState) =                              23
      (TextIO.output(!fileID,observeStr); ())                        24
                                                                     25
   fun init (currentState) =                                         26
      ((fileID := TextIO.openOut("ReceiverLog"));                    27
       currentState)                                                 28
   fun conclude (currentState) =                                     29
      (TextIO.closeOut(!fileID); currentState)                       30
   val init_mode = AtSimStart                                        31
   val conclude_mode = AfterSim                                      32
                                                                     33
   fun monitor(anEvent, currentState) =                              34
      if check (anEvent, currentState)                               35
          then act (observe (anEvent, currentState),                 36
                    currentState)                                    37
      else ()                                                        38
  end                                                                39
```

Figure 8.7: Code for log-file monitor.

Defining the log-file monitor is fairly easy in Design/CPN. The complete code for the log-file monitor can be seen in Fig. 8.7, and a description of how the

code was generated follows. After the user selected the Receive Packet transition, Design/CPN generated template code for the `check` and `observe` functions for the log-file monitor, and the template code that was generated is equivalent to lines 11 and 13 in Fig. 8.7. Since the user selected only the Receive Packet transition, it is implicitly assumed in Design/CPN that the `check` function returns false after all other events in the system. As a consequence, the `check` function should always return true, since the log file should always be updated when a packet is received, regardless of the particular values of the sequence number, the payload, or the receiver's counter. Therefore, the user does not need to make any changes to the template code for the `check` function. The body of the `observe` function is the only part of this monitor that the user had to define. In this case, the user only had to write a few lines of SML code (lines 14-19 in Fig. 8.7) in order to generate a file that resembles the excerpt shown in Fig. 8.2.

After the user defined the `check` and `observe` functions, Design/CPN checked the syntax of the functions and automatically generated the code that is shown in Fig. 8.7. Note that the user never needs to see more than the `check` and `observe` functions. The user chose to name the monitor `ReceiverLog`. The type `event` for this monitor corresponds to the occurrences of the event Receive Packet. The monitor does not explicitly examine any portion of the state of the model, therefore the type `state` is a trivial data type. However, the monitor will implicitly examine some of the state, e.g. the value of the counter in the receiver, but this information is available in the specification of the event Receive Packet. A log-file monitor cannot be used to change the state of the model or the simulator, therefore the `actState` is also a trivial data type. The `observeType` for the monitor will be `string`, since the `observe` function must generate a string which will be saved in the file. The log file will be opened at the start of the simulation (`AtSimStart`) when the `init` function is called by the simulator. Similarly, the file is closed by the `conclude` function at the end of a simulation (`AfterSim`).

The monitor will be invoked only when a packet is received by the receiver. This is achieved by augmenting the implementation of the event Receive Packet to call the `monitor` function for the log-file monitor. As mentioned previously, the `check` function should always return true (because it will only be evaluated when a packet is received), the `observe` function will return a string that is dependent on the contents of the packet, and the predefined `act` function will save the string in a file.

8.4.2 Standard and User-Defined Monitors

Based on the monitoring framework presented in Sect. 8.3, it is possible to construct many different types of monitors. More specifically, it is possible to construct all of the different types of monitors that were discussed in Sect. 8.2. At first glance, it appears that many of the monitors are dependent on the model that they inspect. This would seem to indicate that it would be rather time consuming and difficult for users to define and use monitors. However, it turns out that many monitors can be constructed so that they are (relatively)

independent of the specific model. That makes it possible to integrate so-called *standard monitors* into the tool. Using standard monitors the user only needs to specify minor parts of the monitor – the rest of the monitor is predefined.

The example of the log-file monitor for the receiver's log is an excellent example of a standard monitor. By simply selecting a portion of a model, the majority of the code for the monitor can be automatically generated by the tool, and the user only needs to make simple modifications to the code that is generated. For this standard log-file monitor, it is only possible for the user to change the body of the `check` and `observe` functions

There are likely to be other situations in which such a log-file monitor is insufficient or too rigidly defined. In the example above, it is impossible to read from the file in question. Therefore, it would be helpful for a tool to support a variety of monitors with varying degrees of flexibility. For example, support could be provided for file access monitors that only read or only write strings in files, that record the sequence of events that occur in a simulation, or that observe the state of model and open a file for reading when certain conditions are fulfilled.

In other cases it may be useful to be able to create new types of monitors or monitors that are completely model-dependent. Therefore, it is also useful that the user can create monitors from scratch. If users create monitors that conform to the interface shown in Fig. 8.5, it should be relatively easy for tool developers to incorporate the new monitors into the tool or to create libraries of monitors that can be shared among users, thus extending the usability and flexibility of a simulation tool.

8.5 Conclusion and Future Work

In this paper we have presented a framework for monitoring simulations of discrete-event systems. The purpose of the monitoring framework is to serve as a reference for implementing different types of monitors in discrete-event system modelling tools. The reason for developing this monitoring framework is that most monitors we have seen use the same way to define when to monitor, what to monitor, while only the action based on the monitoring will be different for different monitors. Prototype monitoring facilities based on the framework have been developed and used, and some of the monitors that we have considered are: data collection monitors, message sequence chart monitors, and communication monitors. These examples show that monitors can be used for widely different purposes. This paper is one step in the direction of unifying how monitors are implemented within a given tool.

There are several ideas for future work in this field. In this paper we have only considered monitoring of simulations. However, most of the monitors may also be applied to state spaces. The implication of applying the same monitors to both simulations and state spaces is twofold. Consider a data collection monitor to be used to collect data during simulations. It cannot only be applied to simulations, it can also be applied to paths in the state space to collect data from the entire state space of a model. Likewise, property monitors applied to

state spaces may also be applied on simulations. The implication is that early model checking or simple analysis of a model can be conducted during initial simulations of a model – during the modelling phase – without constructing state spaces. After completing a model, the same monitors can be applied to the complete state space. The consequence is that time is saved by using the same monitors for both simulation and state space analysis.

Chapter 9

Annotating Coloured Petri Nets

The paper "Annotating Coloured Petri Nets" presented in this chapter has been been accepted for presentation at the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'02) [89].

[89] B. Lindstrøm and L. Wells. Annotating coloured Petri nets. To appear in the proceedings of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'02), 2002.

This chapter is, except for minor typographical changes, the same as the paper [89].

Annotating Coloured Petri Nets

Bo Lindstrøm* Lisa Wells*

Abstract

Coloured Petri nets (CP-nets) can be used for several fundamentally different purposes like functional analysis, performance analysis, and visualisation. To be able to use the corresponding tool extensions and libraries it is sometimes necessary to include extra auxiliary information in the CP-net. An example of such auxiliary information is a counter which is associated with a token to be able to do performance analysis. Modifying colour sets and arc inscriptions in a CP-net to support a specific use may lead to creation of several slightly different CP-nets – only to support the different uses of the same basic CP-net. One solution to this problem is that the auxiliary information is not integrated into colour sets and arc inscriptions of a CP-net, but is kept separately. This makes it easy to disable this auxiliary information if a CP-net is to be used for another purpose. This paper proposes a method which makes it possible to associate auxiliary information, called annotations, with tokens without modifying the colour sets of the CP-net. Annotations are pieces of information that are not essential for determining the behaviour of the system being modelled, but are rather added to support a certain use of the CP-net. We define the semantics of annotations by describing a translation from a CP-net and the corresponding annotation layers to another CP-net where the annotations are an integrated part of the CP-net.

9.1 Introduction

Coloured Petri nets (CP-nets or CPNs) were formulated by Kurt Jensen [70, 71] with the primary purpose of specifying, designing, and analysing concurrent systems. The tools Design/CPN [31, 40] and CPN Tools [37] have been developed to give tool-support for creating and analysing CP-nets. Ongoing practical use of CP-nets and Design/CPN in industrial projects [72] have identified the need for additional facilities in the tools.

One industrial project described in [26] illustrated that CP-nets can be used for performance analysis by predicting the performance of a web server using a CPN model. As part of this project, the Design/CPN Performance Tool [87] was developed as an integrated tool extension supporting data collection during simulations. Later, work was done to extend and generalise these data collection facilities to serve as a basis for a common so-called monitoring framework [90].

*Department of Computer Science, University of Aarhus, Åbogade 34, 8200 Århus N, Denmark. E-mail: blind,wells@daimi.au.dk.

Other projects have shown that visualisation of behaviour using so-called message sequence charts (MSCs) [67] is very useful in combination with CP-nets. As a consequence, a library [92] has been developed for creating MSCs during simulations. Other similar libraries are Mimic [106], which is used for visualisation, and Comms/CPN [49], which is used for communicating with external processes.

The fact that a CPN model can be used for several fundamentally different purposes like functional analysis, performance analysis, and visualisation means that it is desirable that the tool extensions and libraries can be used without having to modify the CPN model itself. It should be possible to use a CPN model, for e.g. performance analysis, without having to add extra places, transitions, and colour sets purely for the purpose of collecting data. Optimally, the auxiliary information should not be integrated into colour set and arc inscriptions of a CPN model, but should be kept separately, so that it is easy to disable this information if the CPN model is to be used for something else.

Up to this point it has only been partially possible to use a CPN model for different purposes without having to change the CPN model itself. With the current tools, it is indeed possible to do, e.g. performance analysis without adding transitions and places for the sole purpose of doing the performance analysis. Unfortunately however, it is often necessary to add extra information to colour sets and arc inscriptions to hold, e.g. performance-related information such as the time at which a certain event happened.

This paper presents work on separating auxiliary information from a CPN model by proposing a method which makes it possible to associate auxiliary information, called *annotations*, with tokens without modifying the colour sets of the CPN model. Annotations are pieces of information that are not essential for determining the behaviour of the system being modelled, but rather are added to support a certain use of the CPN model. A CP-net that is equipped with annotations is referred to as an *annotated CP-net*. In an annotated CP-net, every token carries a token colour, and some tokens carry both a token colour and an annotation. A token that carries both a colour and an annotation is called an *annotated token*. Just like a token value, an annotation may contain any type of information, and it may be arbitrarily complex.

Annotations are defined in *annotation layers*. Defining annotations in layers makes it possible to make modular definitions of both a CP-net and one or more layers of auxiliary information that can be used for varying purposes. By defining several different layers of annotations, it is possible to maintain several versions of a CP-net and thereby to use the same basic CP-net for various purposes by adding, removing, or combining annotation layers. An advantage of the annotation layers is that they are defined so that they affect the behaviour of the original CP-net in a very limited and predictable way. Every marking of an annotated CP-net is the same as a marking in the original CP-net, if annotations are removed.

In the following, we will assume that the reader is familiar with CP-nets as defined in [70]. The first half of this paper provides an informal introduction to annotations and an example of how annotations can be used in practice. The second half of the paper provides a formal definition of annotations and proof of

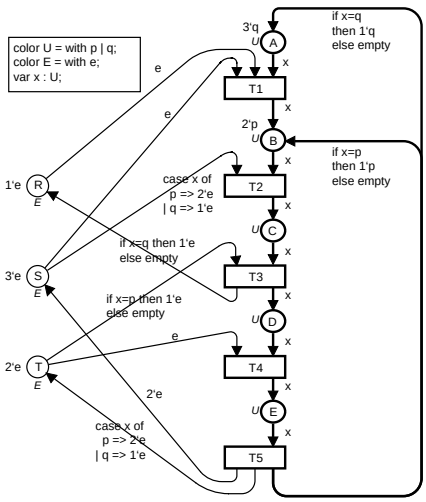


Figure 9.1: The basic CP-net for the resource allocation system.

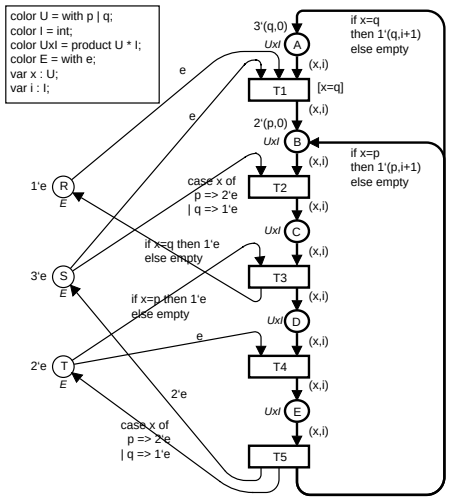


Figure 9.2: The CP-net for the resource allocation system extended with a counter.

the fact that annotations affect the behaviour of a CP-net in a very limited way. In this paper, we will only discuss how to annotate non-hierarchical, untimed CP-nets. However, timed and hierarchical CP-nets can also be annotated using similar techniques.

The paper is structured as follows. Section 9.2 presents the well-known resource allocation system CP-net, which will be used as a running example throughout the paper, and discusses existing ways of including auxiliary information in CP-nets. In Sect. 9.3 we informally introduce our proposal for how to annotate CP-nets. Section 9.4 discusses how multiple annotation layers can be used for visualisation using MSCs. In Sect. 9.5 we give the formal definitions for annotating CP-nets. Finally, in Sect. 9.6 we conclude and give directions for future work.

9.2 Motivation

It is seldom the case that the exact same CP-net can be used for a variety of different purposes, as it is frequently necessary to make small modifications to a CP-net in order to obtain a CP-net that is appropriate for a given purpose. Consider for example the resource allocation system that is found in Jensen's volumes on CP-nets [70, 71]. At least three variations of the resource allocation CP-net can be found in these volumes: a *basic* version (shown in Fig. 9.1) suitable for full state space analysis; an *extended* version (shown in Fig. 9.2) which is extended with cycle counters for the `p` and `q` processes; and a *timed* version with cycle counters and timing information which could be used for performance analysis.

The basic version in Fig. 9.1 purely models the basic aspects of the resource allocation system, and thereby only models the parts of the system that are

common for any use of the CP-net. However, even for such a simple system as the resource allocation system, it is indeed necessary to have slightly different versions of the same CP-net in order to support different kinds of use. In other words, modifications of the basic CP-net are made only to support a certain use, and the modifications may limit other uses of the modified CP-net because the modifications may change the behaviour.

An example of a situation where the basic version does not contain sufficient information is when we need to be able to count how many cycles each of the `p` and `q` processes make in the resource allocation system. The extended CP-net in Fig. 9.2 shows how the basic CP-net can be extended with such auxiliary information. First of all, the colour set `U` has been extended to the product colour set `UxI` to include an integer for the counter in every process token. In addition, the arc inscriptions have been modified to pass on and to update the cycle counters. The cycle counters for the `p` and `q` processes are increased each time a `p` or `q` token passes the transition `T5`. The initial marking has also been modified to include the initial values of the cycle counters. Using this extended CP-net it is possible to determine the number of cycles a process has completed by inspecting the counter of the corresponding tokens.

The version extended with the cycle counters is not useful for all kinds of analysis. This is due to the fact that the cycle counters for the `p` and `q` processes increase each time the `p` and `q` tokens pass transition `T5`, thus resulting in an infinite state space. Therefore, the extended version with cycle counters may be inappropriate for certain kinds of state space analysis. The effect of the cycle counters on the state space can be factored out using equivalence classes, however, it may be annoying to have to remember to manually take care of such auxiliary information before doing state space analysis. In contrast, the state space for the basic CP-net without the cycle counters is finite. This means that the full state space can be generated and analysed, e.g. to prove that the system never reaches a deadlocked state.

Analysing the performance of the resource allocation system is another kind of analysis that requires auxiliary information to be maintained for the tokens in the CP-net. The timed CP-net from [71] could be used to measure the average processing times for each of the two processes. This timed CP-net can be created by modifying the CP-net in Fig. 9.2 by changing the colour set `UxI` to a timed colour set, and by adding an auxiliary component to the colour set to be used for recording the time when a process restarts a cycle,[1] i.e the time at which a `q` process is removed from place `A` or the time at which a `p` process is removed from place `B`. This value can then be used to calculate the processing time for a given process when it passes the `T5` transition. If the timed CP-net should be used for a purpose where the auxiliary information should be ignored, it should often be removed. In the tool Design/CPN, it is easy to disable time, i.e. to consider a timed CP-net as an untimed CP-net. However, auxiliary components that have been added to the colour sets also need to be removed by manually modifying the colour sets and arc inscriptions.

[1]The colour set `U` could be modified to consist of pairs (u,t) where u∈U is a process and t∈TIME is the time at which the process started processing.

From the examples presented above it should now be clear that when using CP-nets for different purposes it is often necessary to maintain different versions of a CP-net with slightly different behaviour. The reason for maintaining different versions is, as mentioned, that it may be necessary to be able to include auxiliary information in tokens. However, the auxiliary information may be extraneous or even disastrous for other uses, e.g. consider the effects of the cycle counters on the size of the state space. Including extra information in a CP-net often requires modification of colour sets, arc expressions, and initialisation expressions.

9.3 Informal Introduction to Annotated CP-nets

In this section we will informally present a method for augmenting tokens in a CP-net with extra or auxiliary information that affects the behaviour of the CP-net in a very limited and predictable manner. To do this we introduce the concept of an *annotation* which is very similar to a token colour in that an annotation is an additional data value that can be attached to a token. An *annotation layer* is used to define annotations and how these annotations are to be associated with tokens in a particular CP-net. An annotation layer cannot be defined independently from a specific CP-net. Therefore, it is always well-defined to refer to the unique CP-net for which an annotation layer is defined. We will refer to this unique CP-net as the *underlying* CP-net of an annotation layer. An *annotated CP-net* is a pair consisting of an annotation layer and its underlying CP-net. We define the semantics of annotations by describing a translation from an annotated CP-net to a CP-net without annotations, referred to as the *matching CP-net*. In practice, the annotations are integrated into the matching CP-net when the translation is made. Section 9.3.1 gives an informal introduction to annotations and annotation layers. Section 9.3.2 describes the intuition of how to translate an annotated CP-net to a matching CP-net. Section 9.3.3 discusses the behaviour of the matching CP-net, and it discusses how the behaviour of the matching CP-net is similar to the behaviour of the underlying CP-net. The formal definition of annotated CP-nets follows in Sect. 9.5.

9.3.1 Annotation Layer

To get an intuitive understanding of how annotations can be used, let us see how the cycle counters that were discussed in Sect. 9.2 can be added as annotations. Recall that Fig. 9.2 shows how the CP-net from Fig. 9.1 can be modified to include the cycle counters as part of the token colours.

Figure 9.3 contains an annotated CP-net for the basic CP-net for the resource allocation system from Fig. 9.1. In Fig. 9.3 the elements from the annotation layer are shown in black, whereas the underlying CP-net is shown in grey. The annotation layer contains *auxiliary declarations* and *auxiliary net inscriptions*, where the auxiliary net inscriptions consist of auxiliary arc expressions, auxiliary colour sets, and auxiliary initialisation expressions.

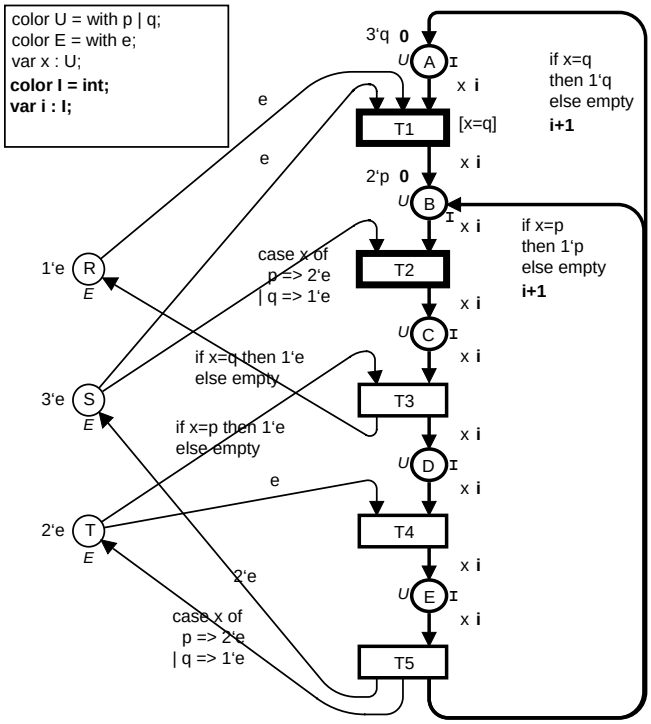


Figure 9.3: Annotated CP-net for the resource allocation system.

The colour set `I` is declared in the first line of the auxiliary declarations. Places `A`, `B`, `C`, `D` and `E` have auxiliary colour set `I` which means that tokens on these places will be annotated tokens that carry integer annotations. A token that carries the annotation `n` has completed the cycle `n` times. Places with auxiliary colour sets are called *annotated places*. The token value for a token on an annotated place has both a token colour and an annotation. Not all places will contain annotated tokens, therefore, some places will not have an associated auxiliary colour set.

All of the annotated places that have an initialisation expression in the underlying CP-net must also have an *auxiliary initialisation expression* in the annotation layer. Places `A` and `B` have the auxiliary initial expression `0`. This expression means that all tokens on places `A` and `B` will have annotation `0` in the initial marking.

All arcs that are connected to annotated places have an *auxiliary arc expression*. In Fig. 9.3, most auxiliary arc expressions consist of the variable `i` which has type `I`. Variable `i` is declared in the annotation layer, therefore, it may only be used within the annotation layer, i.e. it cannot be used in the underlying CP-net. In contrast, variables, colour sets, functions, etc. that are declared in the underlying CP-net may be used both in the underlying CP-net and in the annotation layer. However, certain conditions must be fulfilled in order to ensure that using the same elements in both the annotation layer and the underlying CP-net does not affect the behaviour of the underlying CP-net.

These conditions will be discussed further in Sect. 9.5.2.

Let us consider the intuition behind the auxiliary arc expressions on the arcs surrounding transition `T5`. In the underlying CP-net, `T5` can occur whenever there is one token on place `E`, and this must still be true in an annotated version of the CP-net. Informally, the interpretation of the two types of arc expressions surrounding `T5` is that when transition `T5` occurs with, e.g. binding <x=q, i=5>, one token with colour `q` and annotation `5` will be removed from place `E`. One token with colour `q` and annotation `5+1=6` will be added to place `A`, and the empty multi-set of annotated tokens will be added to place `B`. On the other hand, if `T5` occurs with binding <x=p, i=3>, then one token with colour `p` and annotation `3+1=4` will be added to place `B`, and no tokens will be added to place `A`. In both bindings, multi-sets of (non-annotated) `e` tokens are also added to places `T` and `S`, which are non-annotated places.

The intuition behind the auxiliary inscriptions that have been discussed until now is fairly straightforward. There are, however, some restrictions on the kinds of auxiliary arc expressions that are allowed in order to ensure that annotations have only limited influence on the behaviour of the underlying CP-net. All of the auxiliary arc expressions on arcs from annotated places to transitions consist only of variables, and this is not accidental. For example, the auxiliary arc expression `i` on the arc from `C` to `T3` must not be replaced with, e.g. the constant `4`, which would require that when removing a token from `C` the annotation must be `4`. Allowing such an auxiliary arc expression would mean that the behaviour of the matching CP-net and the underlying CP-net would no longer be similar. Sections 9.5.2 and 9.5.3 discuss the restrictions about which kinds of auxiliary arc expressions are allowed.

9.3.2 Translating an Annotated CP-net to a Matching CP-net

Rather than defining the semantics for annotated CP-nets, we will define the semantics of annotations by describing how an annotated CP-net can be translated to an ordinary CP-net, which is referred to as the *matching* CP-net. The discussion above should have provided a sense of what kinds of annotations the tokens should have and of how an annotated CP-net for the resource allocation system should behave. The annotation layer (referred to as $\mathcal{A}$ and shown in black in Fig. 9.3) and the underlying CP-net (referred to as CPN and shown in Fig. 9.1) constitute an annotated CP-net for the resource allocation system. In this section, we will show how the various auxiliary inscriptions from $\mathcal{A}$ and the inscriptions from CPN are translated to inscriptions in the matching CP-net (referred to as CPN* and shown in Fig. 9.4). The general rules for translating an arbitrary annotation layer and its underlying CP-net are presented in Sect. 9.5.3. In the following, we shall say that a place/arc is annotated/non-annotated in a matching CP-net (like Fig. 9.4) if it is annotated/non-annotated in the corresponding annotated CP-net (like Fig. 9.3).

The rules are simple for translating an annotated CP-net to a matching CP-net. CPN and CPN* have the same net structure. The colour sets for non-annotated places in CPN* are unchanged with respect to CPN; in the example, the colour sets for places `R`, `S` and `T` are unchanged. The colour sets for the

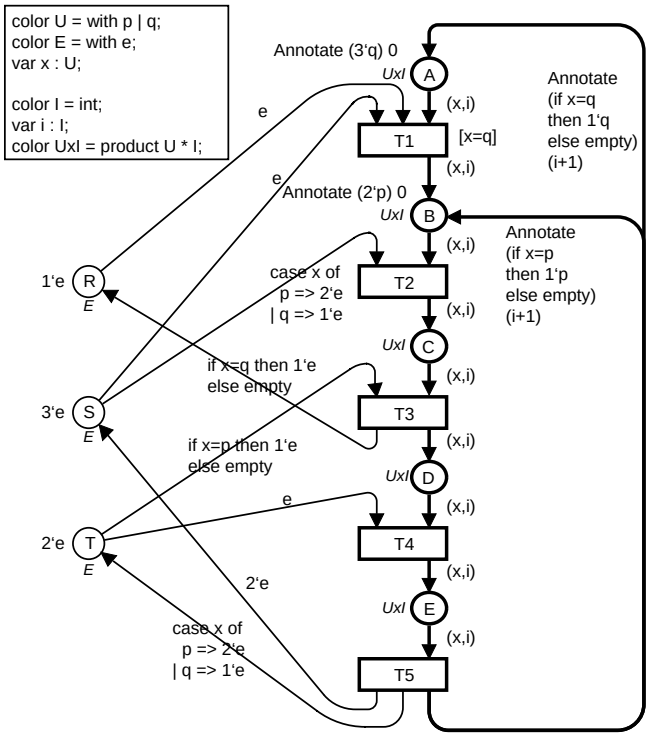


Figure 9.4: Matching CP-net for the resource allocation system.

annotated places are now product colour sets which are products of the original colour sets and the auxiliary colour sets. The colour set for annotated places `A-E` in CPN* is `UxI` which is a product of colour set `U` (the colour set of places `A-E` in CPN) and auxiliary colour set `I` (the auxiliary colour sets from $\mathcal{A}$). The tokens on an annotated place p in CPN* are said to have an annotated colour `(c,a)`, where `c` is a colour (from the colour set of p in CPN), and `a` is an annotation (from the auxiliary colour set of p in $\mathcal{A}$). The set of colour sets for CPN* is the union of the set of colour sets from CPN, the set of auxiliary colour sets from $\mathcal{A}$, and the set of product colour sets for annotated places.

In the previous section, the intuitive meaning of several auxiliary expressions was that a given annotation should be added to all elements in a multi-set of colours. This is the meaning of, for example, all of the auxiliary initial expressions. Let us define a function `Annotate` that, given an arbitrary multi-set and an arbitrary annotation, will annotate all of the elements in the multi-set with the annotation. This function is used in several net inscriptions in CPN*.

Let us consider how the initialisation expressions are created for CPN*. The initialisation expressions for non-annotated places (`R`, `S` and `T`) are unchanged, and evaluating these expressions yields non-annotated multi-sets (`1`e,3`e`, and `2`e` respectively). The initial markings for annotated places in CPN* must be multi-sets of annotated colours. Place `A` has the initialisation expression `Annotate (3`q) 0`, which evaluates to `3`(q,0)` tokens which correspond to the desired multi-set of three annotated tokens, each with colour `q` and annotation

`0`. Note, in particular, that if the annotations are removed from the multi-set `3`(q,0)`, then we obtain the multi-set `3`q` which is exactly the multi-set that is obtained when evaluating the initialisation expression for `A` in the underlying CP-net. Similarly, place `B` has an initial marking of two tokens, each with colour `p` and annotation `0`. The initial markings of the remaining places are empty.

The arc expressions of CPN and the auxiliary arc expressions of $\mathcal{A}$ are combined in a similar manner to create arc expressions for CPN*. If the type of an arc expression in CPN, `expr`, is a single colour, then the arc expression in CPN* is the pair `(expr, aexpr)`, where `aexpr` is the auxiliary arc expression in $\mathcal{A}$. The arc expressions for most annotated arcs in Fig. 9.4 have this form. When the type of an arc expression is a multi-set of colours, then the arc expression for CPN* is `Annotate expr aexpr`. The arc expressions for the arcs from transition `T5` to places `A` and `B` were created in this manner. The next section discusses how the behaviour of the matching CP-net is similar to the behaviour of the underlying CP-net.

9.3.3 Behaviour of Matching CP-nets

In a matching CP-net some places contain annotated tokens, other places contain non-annotated tokens, and occurrences of binding elements can remove and add both regular, non-annotated tokens and annotated tokens. Figure 9.5 shows a marking of the matching CP-net, CPN*. The marking of place `A` contains two tokens – one with colour `(q,4)`, the other with colour `(q,5)`. This corresponds to a marking in the annotated CP-net of Fig. 9.3 where `A` has one token with colour `q` and annotation `4` and another token with colour `q` and annotation `5`. Similarly, place `B` contains three tokens with colours `(p,5)`, `(p,6)` and `(q,1)`. This corresponds to a marking in the annotated net where `B` has one token with colour `p` and annotation `5`, another token with colour `p` and annotation `6`, and a third token with colour `q` and annotation `1`. Finally, places `S` and `T` each contain two non-annotated tokens with colour `e`.

We say that the behaviour of the matching CP-net *matches* the behaviour of its underlying CP-net. Informally this means that every occurrence sequence in the matching CP-net is also an occurrence sequence in the underlying CP-net if annotations are ignored. Furthermore, for every occurrence sequence in the underlying CP-net, it is possible to find at least one matching occurrence sequence in the matching CP-net which is identical to the occurrence sequence from the underlying CP-net when annotations are ignored. Consider, for example, a marking, M, of the basic CP-net from Fig. 9.1, in which there are two `e` tokens on places `S` and `T`, two `q` tokens on place `A`, and two `p` tokens and one `q` token on place `B`. The binding element (`T2`, ⟨x=p⟩) is enabled in M. The marking of the matching CP-net in Fig. 9.5 is the same as marking M if annotations are ignored. The occurrence of either (`T2`, ⟨x=p, i=5⟩) or (`T2`, ⟨x=p, i=6⟩) in the matching CP-net will result in markings that are equal M′ where M[(`T2`, ⟨x=p⟩)⟩M′ when annotations are ignored. A formal definition of matching behaviour can be found in Sect. 9.5.4.

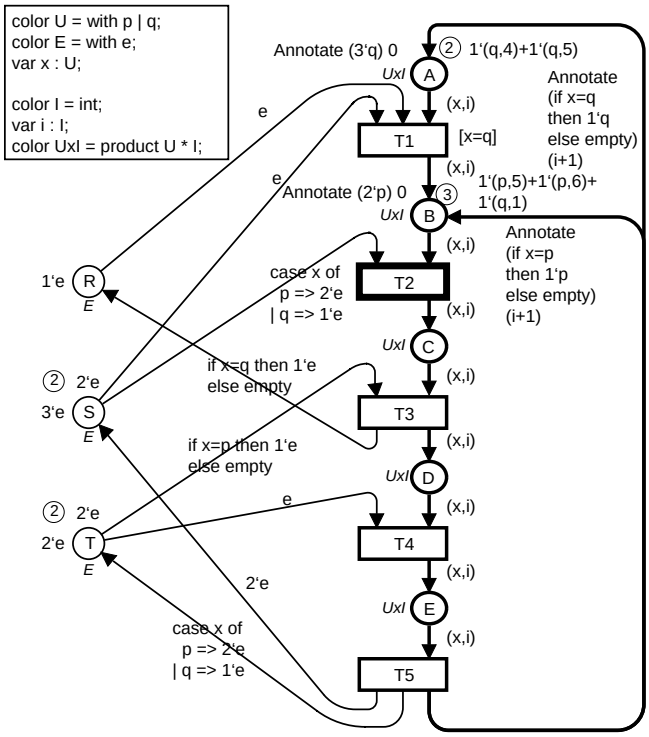


Figure 9.5: Marking of the matching CP-net for the resource allocation system.

9.4 Using Annotation Layers in Practice

This section discusses how to use several annotation layers for the basic CP-net of the resource allocation system presented in Fig. 9.1 in Sect. 9.2. The purpose of this section is to illustrate that multiple annotation layers can be added on top of each other without changing the original model, and to illustrate some of the uses of annotations. We will discuss an example of how annotations can be used for visualising simulation results. In particular we will consider how message sequence charts (MSCs) can be created using annotations.

A MSC can be used, e.g. to visualise the use of resources. Figure 9.6 depicts a MSC for the basic CP-net of the resource allocation system. The MSC contains two vertical lines which represent the activities of allocating and deallocating resources in the resource allocation system. An arrow represents the dependency between the allocation and deallocation of an `S` or an `R` resource.

The MSC in Fig. 9.6 visualises a sequence of allocations of resources by the `p` and `q` processes. The arrows for `p` processes are dashed. The MSC shows that first the `q` process makes a full cycle where it first allocates an `R` and an `S` resource when `T1` occurs, and an additional `S` resource when `T2` occurs. The `R` resource is deallocated when `T3` occurs, and the two `S` resources are deallocated when `T5` occurs. The last five arrows show a situation where two `p` processes interleave with a `q` process. First the `q` process allocates an `R` and an `S` resource, but then two cycles of `p` processes appear (the two dashed arrows)

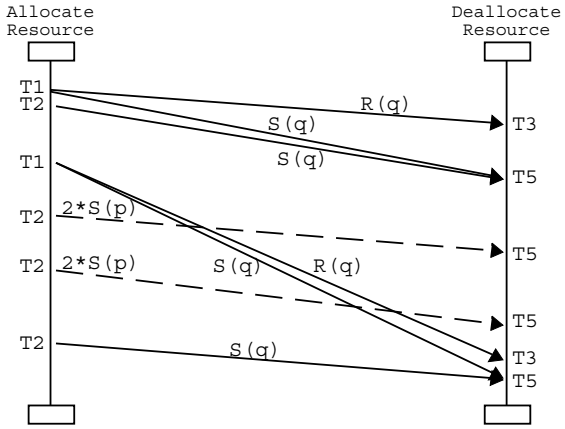


Figure 9.6: Message sequence chart for the resource allocation system.

before the `q` process continues the cycle. This interleaving is explicitly visualised by the arrows started by `T1` and ended by `T3` and `T5`, and crossing the arrows representing the two `p` processes.

A MSC can be generated automatically from a simulation of a CP-net. However, first it is necessary to specify which occurrences of binding elements in the CP-net should generate which arrows in the MSC. Normally, an arrow in a MSC is created when a single transition occurs. However, arrows as illustrated above correspond to two events: one for creating the start-point and one for creating the end-point of an arrow. Such arrows are defined by means of these two points. First the start-point of the arrow is given. Then some other events may appear, and then the event leading to ending the arrow is given. For CP-nets this means that the occurrence of one transition may define the start-point of an arrow while the occurrence of another transition may define the end-point of an arrow. We call such arrows *two-event arrows*.

When using two-event arrows, it is often necessary to annotate a token to hold information of which arrows have been started but not ended yet. In other words, a token must hold the arrow-id of the start-point of the arrow when the first transition occurs and keep it until a transition supposed to end the arrow consumes the token. To avoid modifying the colours of the the CP-net, annotations can be used. Figure 9.7 depicts how the basic CP-net for the resource allocation system can be annotated to generate the MSC in Fig. 9.6. The contents of the annotation layer are shown in black, while the underlying CP-net is shown in grey. The annotation colour `MSC` is an integer which has the purpose of holding the start-point id of an arrow, while the annotation colour `MSCs` is a list of MSC ids. We do not give the details of the functions `msc_start` and `msc_stop` here, however, they are used to set the start-point and end-point of each arrow. In addition each of the functions return a list of arrow-ids of the non-stopped arrows. This list becomes an annotation for the underlying colour. As this example illustrates, we allow auxiliary arc expressions to have side effects. However, the side effects may not affect the behaviour of the underlying CP-net.

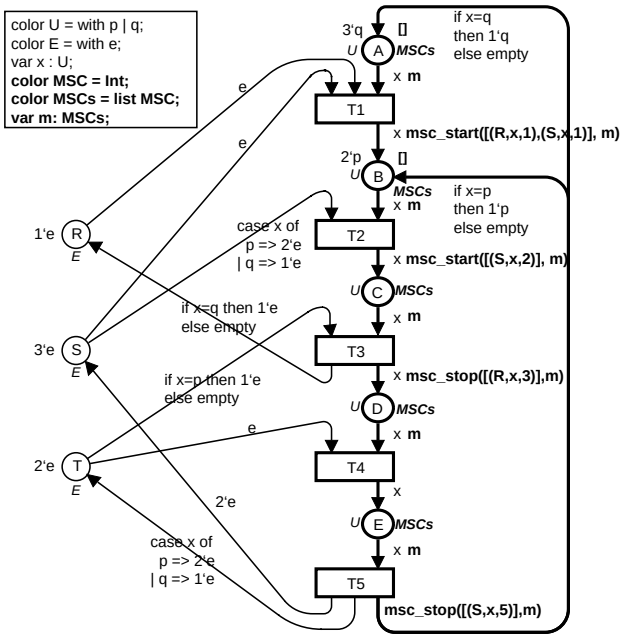


Figure 9.7: Resource allocation system with MSC annotation layer.

The annotation layer in Fig. 9.3 from Sect. 9.3.1 adds a cycle counter to the CP-net. The value of the cycle counter could be included on the arrows in Fig. 9.6 in addition to the type of resource and process. To obtain this, an annotation layer that resembles the MSC Annotation Layer in Fig. 9.7 can be added on top of the cycle counter annotation layer in Fig. 9.3. We will refer to this new annotation layer as Cycle MSC Annotation Layer. In the Cycle MSC Annotation Layer it is possible to refer to the annotations of the Cycle Counter Annotation Layer from the MSC Annotation Layer.

Figure 9.8 depicts some of the possible ways to add annotation layers on top of each other. Notice that an alternative to adding the Cycle MSC Annotation Layer on top of the Cycle Counter Annotation Layer, is to add the original MSC Annotation Layer on top of the Cycle Counter Annotation Layer. This makes sense even though the annotations in Cycle Counter Annotation Layer are not used in the MSC Annotation Layer.

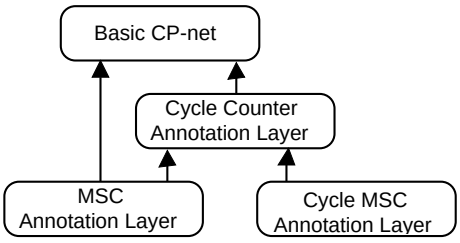


Figure 9.8: Structure of annotation layers for a basic CP-net.

If we had not been able to use an annotation layer for creating the MSC, we would have had to create a new CP-net by adding and modifying the colours of the basic CP-net. For example, the colour sets of the places `A`, `B`,`C`, `D`, and `E` should also hold the `MSCs` colour set. In addition, so-called code segments possibly had to be added to execute the the `msc_start` and `msc_stop` function calls, and the arc-expressions had to be modified to include the MSC variable `m`. In other words, we had to modify the CP-net model itself to generate the MSCs. If the information for updating MSCs is included directly in the CP-net, then it would be difficult to disable the updating of the MSCs, and there is no guarantee that the modifications would not affect the behaviour of the underlying CP-net in unexpected ways.

9.5 Formal Definition of Annotated CP-nets

In this section, we will formally define annotated CP-nets. We will start by introducing some new terminology. We will then define annotation layers, and we will discuss how an annotation layer and a CP-net can be translated into a matching CP-net. We want to define the annotation rules so that they are straightforward to use and understand. To achieve this it turns out to be convenient only to allow annotation of those input arcs of transitions where the arc expressions are uniform with multiplicity one (i.e. always evaluate to a single token colour). For output arcs there are no similar restrictions. If an input arc expression is uniform with multiplicity larger than one, it is usually easy to split the arc into a number of arcs that each have multiplicity one. The requirement can be formally expressed as:

Requirement 1. Let CPN=(Σ, P, T, A, N, C, G, E, I) be a CP-net as defined in Def. 2.5 in [70]. Let $P_{\mathcal{A}} \subseteq P$ be the set of *annotated places*. The following must hold in order to be able to annotate CPN:

$\forall\ p \in P_{\mathcal{A}}$: $\forall\ a \in A$ such that N(a)=(p,t), E(a) must be uniform with multiplicity 1.

This requirement may seem very restrictive. However, in our experience, the kinds of arc inscriptions that are currently not possible to annotate are rarely used in practice. Therefore, the definitions presented here should prove to be useful for annotating many of the CP-nets that are used in practice.

Section 9.5.1 introduces terminology regarding multi-sets of annotated colours. Section 9.5.2 defines annotation layers by describing the auxiliary net inscriptions that are allowed in the annotation layer. Section 9.5.3 presents rules for translating a CP-net and an annotation layer into the matching CP-net, and it discusses the relationship between markings, binding elements, and steps in a matching CP-net and its underlying CP-net. Section 9.5.4 defines matching behaviour. Finally, Sect. 9.5.5 discusses the use of multiple annotation layers.

9.5.1 Multi-sets of Annotated Colours

In the previous section we used expressions such as: `1`(p,5)+1`(p,6)+1`(q,1)` to denote the marking of annotated places[2] in a matching CP-net. This indicates that the marking consists of three tokens with colours `(p,5)`, `(p,6)` and `(q,1)`. However, within the context of annotated CP-nets, this marking can also be interpreted to represent a multi-set of annotated tokens: two tokens with colour `p` and annotations `5` and `6`, and one token with colour `q` and annotation `1`. Multi-sets of annotated elements are ordinary multi-sets[3] of so-called *annotated elements*.

Definition 1. For a non-empty set of elements, S, and a non-empty set of annotations, AN, an *annotated element* (from S) is a pair (s,a), where s∈S and a∈AN. $\pi((s,a))=s$ is the projection of the annotated element (s,a) onto the non-annotated element s.

A *multi-set of annotated elements* over S×AN is a multi-set over S×AN.

If `am` is a multi-set of annotated elements (of S), then `am` *determines* an ordinary (non-annotated) multi-set $\mathtt{am}_\pi$ over S, where $\mathtt{am}_\pi(s) = (\sum_{a \in AN} \mathtt{am}(s,a))\,\mathtt{`}\,s$. $\pi(\mathtt{am})$ is the *projection* of `am` onto the non-annotated multi-set determined by `am`.

If `am` is multi-set over S×AN, `m` is a multi-set over S, and $\pi(\mathtt{am})=\mathtt{m}$, then `am` is said to *cover* `m`, and we say that `m` is *covered* by `am`.

In Sect. 9.3.2 we informally defined the function `Annotate` that will add a given annotation to all elements in a given multi-set. Let us now formally define `Annotate`.

Definition 2. Given an annotation a∈AN and a multi-set $\mathtt{m}=\sum_{s \in S} \mathtt{m}(s)\mathtt{`}s$ over a set S, the function `Annotate` is defined to be:

$$\mathtt{Annotate\ m\ a} = \sum_{s \in S} \mathtt{m}(s)\mathtt{`}(s,\mathtt{a})$$

which is a multi-set of annotated elements of S, i.e. a multi-set with type $(S \times AN)_{MS}$.

As a consequence of Defs. 1 and 2, $\pi(\mathtt{Annotate\ m\ a}) = \mathtt{m}$, for all multi-sets `m` and all annotations `a`.

9.5.2 Annotation Layer

We are now ready to define an annotation layer. An annotation layer is used solely to determine how to add annotations to tokens for a subset of the places in a CP-net. An annotation layer consists of elements that are similar to their

[2] The first paragraph in Sect. 9.3.2 explains what we mean when we refer to an annotated place in a matching CP-net.

[3] Multi-sets as defined in Def. 2.1 in [70]

counterparts in CP-nets. An annotation layer contains auxiliary net inscriptions, and each auxiliary net inscription is associated with an element of the net structure of the underlying CP-net. When translating an annotation layer and its underlying CP-net to the matching CP-net, these auxiliary net expressions will be combined with their counterparts from the underlying CP-net to create colour sets, initialisation expressions and arc expressions for the matching CP-net. There are, however, additional requirements for each of the concepts. An explanation of each item in the definition is given immediately below the definition. A similar remark applies for many of the other definitions in this paper.

Definition 3. Let CPN=(Σ, P, T, A, N, C, G, E, I) be a CP-net. An *annotation layer* for CPN is a tuple $\mathcal{A}$=($\Sigma_{\mathcal{A}}$, $P_{\mathcal{A}}$, $A_{\mathcal{A}}$, $C_{\mathcal{A}}$, $E_{\mathcal{A}}$, $I_{\mathcal{A}}$) where

i. $\Sigma_{\mathcal{A}}$ is a finite set of non-empty sets, called *auxiliary colour sets*, where $\Sigma \subseteq \Sigma_{\mathcal{A}}$.

ii. $P_{\mathcal{A}} \subseteq P$ is a finite set of *annotated places*.

iii. $A_{\mathcal{A}} \subseteq A$ is the finite set of *annotated arcs*, where $A_{\mathcal{A}} = A(P_{\mathcal{A}})$.

iv. $C_{\mathcal{A}}$ is an *auxiliary colour* function. It is a function from $P_{\mathcal{A}}$ into $\Sigma_{\mathcal{A}}$.

v. $E_{\mathcal{A}}$ is an *auxiliary arc expression* function. It is defined from $A_{\mathcal{A}}$ into expressions such that:

$$\forall\, a \in A_{\mathcal{A}}\text{: } Type(E_{\mathcal{A}}(a)) = C_{\mathcal{A}}(p(a)) \wedge Type(Var(E_{\mathcal{A}}(a))) \subseteq \Sigma_{\mathcal{A}}$$

vi. $I_{\mathcal{A}}$ is an *auxiliary initialisation* function. It is a function from $P_{\mathcal{A}}$ into closed expressions such that:

$$\forall\, p \in P_{\mathcal{A}}\text{: } Type(I_{\mathcal{A}}(p)) = C_{\mathcal{A}}(p).$$

i. The set of *auxiliary colour sets* is the set of colour sets that determine the types, operations and functions that can be used in the auxiliary net inscriptions. The auxiliary colour sets determine the type of annotations that the tokens on the annotated places carry. All colour sets from the underlying CP-net can be used as auxiliary colour sets. Additional auxiliary colour sets may be declared within an annotation layer.

ii. The set of annotated places are the only places that are allowed to contain annotated tokens.

iii. The annotated arcs are exactly the surrounding arcs for the places in $P_{\mathcal{A}}$.

iv. The *auxiliary colour function*, $C_{\mathcal{A}}$, is a function from $P_{\mathcal{A}}$ into $\Sigma_{\mathcal{A}}$, and is defined analogously to the colour function for CP-nets. Thus, for all $p \in P_{\mathcal{A}}$, $C_{\mathcal{A}}(p)$ is the auxiliary colour set of p.

v. Auxiliary arc expressions are only allowed to evaluate to a single annotation of the correct type. If the arc expression of an arc is missing in CPN, then we require that its auxiliary arc expression is also missing in $\mathcal{A}$.

vi. The auxiliary initialisation function maps each annotated place, p, into a closed expression which must be of type $C_{\mathcal{A}}(p)$, i.e. a single annotation from $C_{\mathcal{A}}(p)$. If the initial expression of place p is missing in CPN, then we require that its auxiliary initial expression is also missing in $\mathcal{A}$.

9.5.3 Translating Annotated CP-nets to Matching CP-nets

We will now define how to translate an annotated CP-net, $(CPN, \mathcal{A})$, to a new CP-net, CPN*, which is called a matching CP-net. CPN* and CPN have the same net structure. Net inscriptions for non-annotated places, non-annotated arcs, and transitions in CPN* are unchanged with respect to CPN. In contrast, net inscriptions for annotated places and annotated arcs in CPN* are obtained by combining net inscriptions from CPN with their counterpart auxiliary net inscriptions in $\mathcal{A}$. A matching CP-net is defined below.

Definition 4. Let $(CPN, \mathcal{A})$ be an annotated CP-net, where CPN=(Σ, P, T, A, N, C, G, E, I) and $\mathcal{A}$=($\Sigma_{\mathcal{A}}$, $P_{\mathcal{A}}$, $A_{\mathcal{A}}$, $C_{\mathcal{A}}$, $E_{\mathcal{A}}$, $I_{\mathcal{A}}$) is a annotation layer. We define the *matching CP-net* to be CPN*=(Σ^*, P*, T*, A*, N*, C*, G*, E*, I*) where

i. $\Sigma^* = \Sigma_{\mathcal{A}} \cup \{C(p) \times C_{\mathcal{A}}(p) \mid p \in P_{\mathcal{A}}\}$.

ii. $P^* = P$

iii. $T^* = T$

iv. $A^* = A$

v. $N^* = N$

vi. $C^*(p) = \begin{cases} C(p) & \text{if } p \notin P_{\mathcal{A}} \\ C(p) \times C_{\mathcal{A}}(p) & \text{if } p \in P_{\mathcal{A}} \end{cases}$

vii. $G^* = G$

viii. $E^*(a) = \begin{cases} E(a) & \text{if } a \notin A_{\mathcal{A}} \\ \text{Annotate } E(a)\ E_{\mathcal{A}}(a) & \text{if } a \in A_{\mathcal{A}} \end{cases}$

ix. $I^*(p) = \begin{cases} I(p) & \text{if } p \notin P_{\mathcal{A}} \\ \text{Annotate } I(p)\ I_{\mathcal{A}}(p) & \text{if } p \in P_{\mathcal{A}} \end{cases}$

i. $\{C(p) \times C_{\mathcal{A}}(p) \mid p \in P_{\mathcal{A}}\}$ is the set of product colour sets for the annotated places in CPN*.

ii. + iii. + iv. + v. The places, transitions, arcs, and node function in CPN* are unchanged with respect to CPN.

vi. Defining the colour function C* is straightforward. The colour set for a non-annotated place in CPN* is the same as its colour set in CPN. The colour set for an annotated place p in CPN* is $C(p) \times C_{\mathcal{A}}(p)$.

vii. The guard function in CPN* is unchanged with respect to CPN.

viii. The arc expression for a non-annotated arc a in CPN* is the same as the arc expression for a in CPN. If the arc expression for a is missing in CPN, then its arc expression will also be missing in CPN*. This is shorthand for empty, as usual for CP-nets. The arc expression for an annotated arc a in CPN* is derived from the arc expression for a in CPN and the auxiliary arc expression for a in $\mathcal{A}$. The expression `Annotate (E(a)) (E`$_{\mathcal{A}}$`(a))` will yield a multi-set with type $(\mathrm{C(p(a))}\times\mathrm{Type(E}_{\mathcal{A}}\mathrm{(p(a))))}_{MS}$ which is exactly $(\mathrm{C(p(a))}\times\mathrm{C}_{\mathcal{A}}\mathrm{(p(a))})_{MS}$, as required. If an arc expression (for an annotated arc) evaluates to a single colour in CPN, then we allow the arc expression for CPN* to be the pair `(E(a), E`$_{\mathcal{A}}$`(a))` where the first element is the arc expression from CPN, and the second element is the auxiliary arc expression from $\mathcal{A}$. This is shorthand for the multi-set `1`` `(E(a), E`$_{\mathcal{A}}$`(a))`.

ix. If p is not an annotated place, then the initial expression of p in CPN* is unchanged with respect to CPN. If the initial expression of a place is missing in CPN, then its initial expression will also be missing in CPN*. A missing initial expression is shorthand for the empty. For an annotated place p, the expression `Annotate (I(a)) (I`$_{\mathcal{A}}$`(a))` will yield a multi-set with type $(\mathrm{C(p(a))}\times\mathrm{Type(I}_{\mathcal{A}}\mathrm{(p(a))))}_{MS}$ which is exactly $(\mathrm{C(p(a))}\times\mathrm{C}_{\mathcal{A}}\mathrm{(p(a))})_{MS}$, as required. $\mathrm{I^*(p)}$ is a closed expression for all p, since I(p) is a closed expression for all p, and $\mathrm{I}_{\mathcal{A}}$(p) is closed for all $\mathrm{p}\in\mathrm{P}_{\mathcal{A}}$. When the type of the initial expression for an annotated place in the underlying CP-net is a single colour, then we allow the the initial expression in CPN* to be the pair `(I(p),I`$_{\mathcal{A}}$`(p))` that is uniquely determined by the initial expression of p in CPN and the auxiliary initial expression of p in $\mathcal{A}$. This is shorthand for `1`` `(I(p),I`$_{\mathcal{A}}$`(p))`.

Covering Markings, Bindings and Steps

We will now define what it means for markings, bindings and steps of a matching CP-net to cover the markings, bindings and steps of its underlying CP-net.

Definition 5. Let (CPN, $\mathcal{A}$) be an annotated CP-net with matching CP-net CPN*. We then define three projection functions π that map a marking M^* of CPN* into a marking M of CPN, a binding b^* of a transition t in CPN* into a binding b of t in CPN, and a step Y^* of CPN* into a step Y of CPN, respectively.

i. $\forall p \in P^*: (\pi(M^*))(p) = \begin{cases} M^*(p) & \text{if } p \notin P_{\mathcal{A}} \\ \pi(M^*(p)) & \text{if } p \in P_{\mathcal{A}} \end{cases}$

ii. $\forall v \in Var(t): (\pi(b^*))(v) = b^*(v)$, where Var(t) are the variables of t in CPN.

iii. $(\pi(Y^*)) = \sum_{(t,b^*) \in Y^*} (Y^*(t,b^*))`(t,\pi(b^*))$

If $\pi(M^*)=M$, $\pi(b^*)=b$, and $\pi(Y^*)=Y$, then we say that M^*, b^*, and Y^* *cover* M, b, and Y, respectively. We also say that M, b, and Y are *covered* by M^*, b^*, and Y^*, respectively.

i. Given a marking of a matching CP-net, π will remove the annotations from the tokens on annotated places, and it will leave the markings of non-annotated places unchanged. A marking of a matching CP-net covers a marking of its underlying CP-net, if the two markings are equal when annotations in the first marking are ignored.

ii. Given a binding of a transition in CPN*, π removes the bindings of the variables in $Var^*(t) \backslash Var(t)$, i.e. π removes the bindings of the variables of t that are not found in CPN. A binding of a transition in CPN* covers a binding of the corresponding transition in CPN when the variables that are found in both CP-nets are bound to the same value.

iii. For each binding element (t,b^*) in Y^*, π removes the bindings of the variables of t that are not found in CPN.

We define similar functions that map the set of markings ($\mathbb{M}^*$), the set of steps ($\mathbb{Y}^*$), the set of token elements (TE^*), and the set of binding elements (BE^*) of CPN* into the corresponding sets in CPN:

$$\pi(\mathbb{M}^*)=\{\pi(M^*): M^* \in \mathbb{M}^*\}$$

$$\pi(TE^*)=\{(p, c^*) \mid p \notin P_{\mathcal{A}} \text{ and } c^* \in C^*(p)\} \cup \{(p, \pi(c^*)) \mid p \in P_{\mathcal{A}} \text{ and } c^* \in C^*(p)\}$$

$$\pi(\mathbb{Y}^*)=\{\pi(Y^*): Y^* \in \mathbb{Y}^*\}$$

$$\pi(BE^*) = \{(t,\pi(b^*)) \mid (t,b^*) \in BE^*\}$$

Sound Annotation Layers

Definition 3 defines the syntax for elements in annotation layers, but it does not guarantee that annotations do not affect the behaviour of the underlying

CP-net. Instead of specifying which kinds of auxiliary arc inscriptions are allowed, we will define a more general property that has to be satisfied.

Definition 6. Let (CPN, $\mathcal{A}$) be an annotated CP-net with matching CP-net CPN*. $\mathcal{A}$ is a *sound* annotation layer if the following property is satisfied:

$\forall M \in \mathbb{M}, \forall Y \in \mathbb{Y}, \forall M^* \in \mathbb{M}^*: M[Y\rangle \wedge \pi(M^*)=M \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M^*[Y^*\rangle$

where $\mathbb{M}$ and $\mathbb{Y}$ are the set of markings and the set of steps, respectively, for CPN, and $\mathbb{M}^*$ and $\mathbb{Y}^*$ are the analogous sets for CPN*.

Assume that the step Y is enabled in the marking M in the underlying CP-net. Let M* be a marking of CPN* that covers M. Definition 6 states that it must be possible to find a step Y* that covers Y, and Y* must be enabled in M*. The soundness of an annotation layer is essential for showing that for every occurrence sequence in the underlying CP-net, there is at least one matching occurrence sequence in the matching CP-net which is identical to the occurrence sequence from the underlying CP-net when annotations are ignored.

The auxiliary arc expressions on input arcs to a transition will be used, in part, to determine if the transition is enabled in a given state of the matching CP-net. By limiting the kinds of auxiliary arc expressions that are allowed on input arcs to transitions, it is possible to guarantee that annotations cannot restrict the enabling of a transition in the matching CP-net with respect to what is allowed in the underlying CP-net. Exactly which kinds of auxiliary arc expressions should be allowed may be decided by the implementors of tools supporting CP-nets. It is also the responsibility of tool implementors to prove that their allowable set of auxiliary arc expressions fulfil Def. 6. An example of an allowable auxiliary arc expression for arc a is a single variable `v`. However, `v` must also fulfil the following: `v` may not found in any arc expressions for the arcs surrounding t(a), and `v` may not be found in any other auxiliary arc expression for input arcs to t(a).

9.5.4 Matching Behaviour

In the previous sections we have stated that the behaviour of a matching CP-net *matches* the behaviour of its underlying CP-net. Informally this means that every occurrence sequence in a matching CP-net corresponds to an occurrence sequence in the underlying CP-net, and for every occurrence sequence in the underlying CP-net, it is possible to find at least one corresponding occurrence sequence in the matching CP-net. If a matching CP-net is derived from a CP-net and a *sound* annotation layer, then the following theorem shows how the behaviour of the matching CP-net matches the behaviour of its underlying CP-net.

Theorem 1. *Let $(CPN, \mathcal{A})$ be an annotated CP-net with a sound annotation layer. Let CPN^* be the matching CP-net derived from $(CPN, \mathcal{A})$. Let M_0, $\mathbb{M}$, and $\mathbb{Y}$ denote the initial marking, the set of all markings, and the set of all steps, respectively, for CPN. Similarly, let M_0^*, $\mathbb{M}^*$, and $\mathbb{Y}^*$ denote the same concepts for CPN^*. Then we have the following properties:*

i. $\pi(\mathbb{M}^*)=\mathbb{M} \wedge \pi(M_0^*)=M_0$.

ii. $\pi(\mathbb{Y}^*)=\mathbb{Y}$.

iii. $\forall\ M_1^*,\ M_2^* \in \mathbb{M}^*,\ \forall\ Y^* \in \mathbb{Y}^*:\ M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle \pi(M_2^*)$

iv. $\forall\ M_1, M_2 \in \mathbb{M},\ \forall\ Y \in \mathbb{Y},\ \forall\ M_1^*, M_2^* \in \mathbb{M}^*:$
$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists\, Y^* \in \mathbb{Y}^*:\ \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$

i. The markings of a matching CP-net cover the markings of its underlying CP-net. The markings of the underlying CP-net are covered by the markings of the matching CP-net. The initial marking of a matching CP-net covers the initial marking of its underlying CP-net.

ii. The steps of a matching CP-net cover the steps of its underlying CP-net. The steps of the underlying CP-net are covered by the steps of the matching CP-net.

iii. An occurrence sequence of length one in the matching CP-net covers an occurrence sequence of length one in its underlying CP-net. In other words, if marking M_2^* is reached by the occurrence of Y^* in marking M_1^* in CPN^*, then $\pi(M_2^*)$ will be reached by the occurrence of $\pi(Y^*)$ in $\pi(M_1^*)$ in CPN.

iv. An occurrence sequence of length one in the underlying CP-net can be covered by an occurrence sequence of length one in the matching CP-net. If M_2 is reached by the occurrence of Y in M_1 in CPN, and if marking M_1^* in CPN^* covers M_1, then it is always possible to find a step Y^* in CPN^*, such that Y^* covers Y and is enabled in M_1^*. If the occurrence of Y^* in M_1^* yields the marking M_2^*, then M_2^*will cover M_2.

The proof for Theorem 1 can be found in Appendix 9.A.

9.5.5 Multiple Annotation Layers

The previous sections have discussed how to create a single annotation layer for a CP-net. The purpose of introducing an annotation layer is to make it possible to separate annotations from the CP-net, and to annotate a CP-net for several different purposes like, e.g. performance analysis and MSCs. However, if only one annotation layer exists, then it is not possible to easily disable, e.g. only the annotations for performance analysis, while still using the annotations for MSCs. The reason is, that all annotations have to be written in the one and only

annotation layer. This motivates the need for multiple layers of annotations. When multiple annotation layers are allowed, then independent annotations can be written in separate annotation layers, and thereby making it easy to enable and disable each of the independent annotation layers.

Definition 7 defines multiple annotation layers. Multiple annotation layers are defined using the fact that a single annotation layer, $\mathcal{A}_1$, and a CP-net, CPN, is translated to another CP-net, CPN_1^*. Seen from another annotation layer, $\mathcal{A}_2$, CPN_1^* is essentially the same as CPN aside from the added annotations, and can therefore be annotated with an annotation layer $\mathcal{A}_2$. The consequence of this definition is that $\mathcal{A}_2$ can refer to annotations in $\mathcal{A}_1$. In general, annotations in annotation layer $\mathcal{A}_i$ can refer to annotations in annotation layer $\mathcal{A}_j$ when $j \leq i$.

Definition 7. Let CPN be a CP-net and let $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_n$ be annotation layers for CPN. Let τ be the translation from an annotation layer $\mathcal{A}$ and a corresponding CP-net CPN to CPN*, as defined in Sect. 9.5.3. Then CPN* with multiple annotation layers is defined by:

$$CPN^* = \tau(\ldots\tau(\tau(\text{CPN},\mathcal{A}_1),\mathcal{A}_2),\mathcal{A}_n)$$

9.6 Conclusion

In this paper we have discussed annotations for CP-nets where annotations are used to add auxiliary information to tokens. Auxiliary information is needed to support different uses of a single CP-net, such as for performance analysis and visualisation, thus the information should not have influence on the dynamic behaviour of a CPN model. One of the advantages of using annotations instead of manually extending the colour sets in a CPN model is that annotations are specified separately from the colour sets and arc inscriptions. That means that it is easy to enable and disable annotations from being part of the simulation. This is a great advantage when using a model for several purposes such as functional analysis, performance analysis, and visualisation. In addition, it is a great advantage that the behaviour of the matching CP-net matches the behaviour of the underlying CP-net in a very specific and predictable way.

Related work is considered in, e.g. Lakos' work on abstraction [83], where behaviour-respecting abstractions of CP-nets have been investigated, and a so-called colour refinement is proposed. This colour refinement is used to specify more detailed behaviour in sub-modules by extending colour sets to larger domains. The refined colours are only visible in the sub-modules, and the refined colours will typically contain information that is necessary for modelling the behaviour of the system in question. This colour refinement somewhat corresponds to our way of extending colour sets by adding annotations to colours. We are not aware of any other work that addresses the problem of introducing *auxiliary* information into a CP-net (or any other type of simulation model) while at the same time preserving the behaviour of the CP-net. Nor do we know of any other method that can be used to automatically enable or dis-

able different kinds of instrumentation when analysing different aspects of one particular model.

Ex*Spect* [122] is another tool for CP-nets. The tool provides libraries of so-called building blocks that provide support for, e.g., creating message sequence charts and performance analysis. Each building block is similar to a substitution transition and its subpage in Design/CPN. In Ex*Spect* all information that is necessary for updating a MSC or for collecting performance data is included in token colours. Reading the relevant data from token values and processing it is also encoded directly into the model via the building blocks. For example, the building block that can be used to calculate performance measures contains a place which holds the current result. When a certain transition occurs, a new value can be read from a binding element, and the result on this place is updated accordingly. While the building blocks are very easy to use, no attempt is made to separate auxiliary information from a CP-net, and the behaviour of the CP-net also reflects behaviour that is probably not found in the system being modelled.

There are many issues that can be addressed in future work regarding annotations. The techniques that have been presented here have not yet been used in practice. Clearly, it is important that support for annotations be implemented in a CPN tool in order to investigate the practicality and usefulness of the proposed method. Future work includes additional research on dealing with arc inscriptions that do not evaluate to a single colour on input arcs to transitions. In addition, further work is required to improve our proposal of how to add annotations to multi-sets of tokens. The definition of annotation layers states that it is only possible to add one particular annotation to all elements in a multi-set that is obtained by evaluating either an initial expression or an arc expression on an output arc from a transition. This is unnecessarily restrictive, and it should be generalised to make it possible to add different annotations to different elements in a multi-set. Practical experience with annotations may also show that the definition of annotation layers should be extended to include the possibility of defining guards in annotation layers.

In this paper we have only considered how to add annotations to existing arcs expressions, and thereby only considered how to annotate existing tokens. However, it might be useful also to be able to add net structure to the annotation layers. As an example, a place could be added only to the annotation layer with a token to hold a counter with the number of occurrences of a transition. Allowing additional net structure at the annotation layers would make it possible to take advantage of the powerfulness of the graphical notation of CP-nets when encoding the logics of the annotations.

We have only discussed separating the auxiliary annotations and the CP-net from each other. This could be generalised to also allow splitting a CP-net into layers where more layers can be combined to specify the full behaviour of a CP-net. In other words, the specification of the behaviour in a CP-net could be split in more layers. As an example, reconsider the resource allocation CP-net in Fig. 9.1 in Sect. 9.2. The loop handling the resource on the place R (R, T1, B, T2, C, and T3) is to some extent independent from the remaining model (even though it has impact on the behaviour). This loop could be separated

from the remaining CPN model into a new layer to emphasise the fact that the loop is an extra requirement that can be added to the system. This facility could turn out to be very useful when a modeller is simplifying a CP-net to, e.g. be able to generate a sufficiently small state space to be able to analyse it. It would be a matter of moving the parts of the net structure that should not be included when generating the state space to another layer, and then only conduct the analysis on the remaining parts of the CP-net. This could be obtained by disabling the layer with the unneeded behaviour, and the state space could be generated. The advantage is that now a single model exists with layers specifying different behaviour which can be enabled or disabled – instead of having several similar models. Finally, such layers can also make it easier to develop tools where more people can work on a model concurrently, when they operate on different layers.

Acknowledgements We would like to thank Kurt Jensen who has read several drafts of this paper and has provided invaluable comments and feedback. We would also like to thank Søren Christensen, Louise Elgaard and Thomas Mailund who have also read and commented on previous drafts of this paper. Finally, we would also like to thank the anonymous reviewers for their comments and suggestions.

9.A Proof of Matching Behaviour

Theorem 1 (same as in Sect. 9.5.4) *Let (CPN, $\mathcal{A}$) be an annotated CP-net with a sound annotation layer. Let CPN* be the matching CP-net derived from (CPN, $\mathcal{A}$). Let M_0, $\mathbb{M}$, and $\mathbb{Y}$ denote the initial marking, the set of all markings, and the set of all steps, respectively, for CPN. Similarly, let M_0^*, $\mathbb{M}^*$, and $\mathbb{Y}^*$ denote the same concepts for CPN*. Then we have the following properties:*

i. $\pi(\mathbb{M}^*)=\mathbb{M} \wedge \pi(M_0^*)=M_0$.

ii. $\pi(\mathbb{Y}^*)=\mathbb{Y}$.

iii. $\forall\, M_1^*, M_2^* \in \mathbb{M}^*, \forall\, Y^* \in \mathbb{Y}^*: M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle \pi(M_2^*)$

iv. $\forall\, M_1, M_2 \in \mathbb{M}, \forall\, Y \in \mathbb{Y}, \forall\, M_1^*, M_2^* \in \mathbb{M}^*:$
$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$

Proof: The proof is a simple consequence of earlier definitions and Jensen's definitions for CP-nets [70]. Let TE, (t,b), and BE denote the set of all token elements, a binding element, and the set of all binding elements, respectively, for CPN. Similarly, let TE^*, (t,b^*), BE^* denote the same concepts for CPN*.

Before showing that the above properties hold, we will show that the following holds for all annotated arcs:

$\forall\, (t,b^*), \forall\, a \in A_{\mathcal{A}} \cap A(t): \pi(E^*(a)\langle b^*\rangle)=E(a)\langle \pi(b^*)\rangle$. (†)

Let (t,b^*) and $a \in A_{\mathcal{A}} \cap A(t)$ be given.

$\pi(E^*(a)\langle b^*\rangle) \overset{Def.\ 4.viii}{=} \pi\big((\text{Annotate } E(a)\ E_{\mathcal{A}}(a))\langle b^*\rangle\big) \overset{Defs.\ 1\&2}{=} E(a)\langle b^*\rangle \overset{Def.\ 5.ii}{=} E(a)\langle\pi(b^*)\rangle$

Property i. We will show that $\mathbb{M}=\pi(\mathbb{M}^*)$. It is straightforward to show that $\pi(\mathbb{M}^*)=(\pi(TE^*))_{MS}$, and the proof is therefore omitted. From Def. 2.7 in [70] we have that $\mathbb{M}=TE_{MS}$. Thus it is sufficient to show that $TE=\pi(TE^*)$. The definition of $\pi(TE^*)$ gives us:

$\pi(TE^*)=\{(p, c^*) \mid p\notin P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\} \cup \{(p, \pi(c^*)) \mid p\in P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\}$

which by the definition of C^* (Def. 4.vi) is equivalent to:

$\pi(TE^*)=\{(p, c) \mid p\notin P_{\mathcal{A}} \text{ and } c\in C(p)\} \cup \{(p, \pi(c^*)) \mid p\in P_{\mathcal{A}} \text{ and } c^*\in C(p)\times C_{\mathcal{A}}(p)\}$

which by the definition of the projection of annotated elements (Def. 1) is equivalent to:

$\pi(TE^*)=\{(p, c) \mid p\notin P_{\mathcal{A}} \text{ and } c\in C(p)\} \cup \{(p, c) \mid p\in P_{\mathcal{A}} \text{ and } c\in C(p)\}$

the two sets can be combined and we have:

$\pi(TE^*)=\{(p, c) \mid p\in P \text{ and } c\in C(p)\} \overset{Def.\ 2.7in\ [70]}{=} TE$

To show that $\pi(M_0^*)=M_0$, we will show that $\forall p\in P^*$: $(\pi(M_0^*))(p)=M_0(p)$. Consider non-annotated places:

$\forall p\notin P_{\mathcal{A}}$: $(\pi(M_0^*))(p) \overset{Def.\ 5.i}{=} M_0^*(p) = I^*(p) \overset{Def.\ 4.ix}{=} I(p) = M_0(p)$

Consider annotated places:

$\forall p\in P_{\mathcal{A}}$: $(\pi(M_0^*))(p) \overset{Def.\ 5.i}{=} \pi(M_0^*(p)) = \pi(I^*(p)) \overset{Def.\ 4.ix}{=} \pi(\text{Annotate } I(p)\ I_{\mathcal{A}}(p)) \overset{Defs.\ 1\&2}{=} I(p) = M_0(p)$

Property ii. We must show that $\mathbb{Y}=\pi(\mathbb{Y}^*)$. It is straightforward to show that $\pi(\mathbb{Y}^*)= (\pi(BE^*))_{MS}$, therefore the proof is omitted. From Def. 2.7 in [70] we have that $\mathbb{Y}=BE_{MS}$, therefore it is sufficient to show that $BE=\pi(BE^*)$, which we will do by showing: $(t,b')\in\pi(BE^*)\Leftrightarrow (t,b')\in BE$.

Let us show $\Rightarrow$: Let $(t, b')\in\pi(BE^*)$ be given. There exists $(t,b^*)\in BE^*$ such that $(t, \pi(b^*))=(t, b')$ (by definition of $\pi(BE^*)$). b^* is a binding of t in CPN*, therefore for all $v\in Var^*(t)$, where $Var^*(t)$ is the set of variables for t in CPN*, $b^*(v)\in Type(v)$, and b^* fulfils the guard of t in CPN*, i.e. $G^*(t)\langle b^*\rangle$.

From Def. 5.ii we have that for all $v\in Var(t)$, $(\pi(b^*))(v)=b^*(v)$, and we know that $b^*(v)\in Type(v)$. Since $G^*(t)=G(t)$ (Def. 4.vii) and $Var(G(t))\subseteq Var(t)$, we can conclude that $G(t)\langle\pi(b^*)\rangle$, i.e. $\pi(b^*)$ fulfills the guard of t in CPN. From the definition of a binding (Def. 2.6 in [70]), we have that $\pi(b^*)$ is a binding for t in CPN, therefore $(t, \pi(b^*))=(t,b')$ is a binding element for CPN, i.e. $(t, b')\in BE$.

Let us show $\Leftarrow$: Let $(t, b')\in BE$ be given. Using arguments that are similar to the above it is straightforward to show that b' fulfills the guard for t in CPN*, i.e. $G^*(t)\langle b'\rangle$. The binding b' does not bind the variables in $Var^*(t)\backslash Var(t)$. Define a new function b^* on $Var^*(t)$:

$$b^*(v) = \begin{cases} b'(v) & \text{if } v \in Var(t) \\ \text{an arbitrary value from Type}(v) & \text{if } v \in Var^*(t) \backslash Var(t) \end{cases}$$

According to Def. 2.6 in [70], b^* is a binding for t in CPN*. Therefore, (t, b^*) is a binding element for CPN*. By definition of b^*, we have that $\pi(b^*)=b'$, and as a result, (t, b')$\in\pi(BE^*)$.

Property iii. We must show that $\forall\ M_1^*,\ M_2^* \in \mathbb{M}^*,\ \forall\ Y^* \in \mathbb{Y}^*$: $M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle\pi(M_2^*)$

We will first show that $\pi(M_1^*)[\pi(Y^*\rangle$. By the *enabling rule* (Def. 2.8 in [70]) we have that:

$$\forall p \in P^*: \sum_{(t,b^*)\in Y^* a\in A(p,t)} E^*(a)\langle b^*\rangle \leq M_1^*(p) \qquad (*)$$

Consider non-annotated places and non-annotated arcs. Since $E^*=E$ for all non-annotated arcs (by Def. 4.viii), and $M_1^*=\pi(M_1^*)$ for all non-annotated places (by Def. 5.i), it follows from $(*)$ that:

$$\forall p \notin P_{\mathcal{A}}: \sum_{(t,b^*)\in Y^* a\in A(p,t)} E(a)\langle b^*\rangle \leq (\pi(M_1^*))(p)$$

which by the fact that $\pi(b^*)=b^*$ for all variables in Var(E(a)) (by Def. 5.ii) and the definition of $\pi(Y^*)$ (Def. 5.iii) is equivalent to:

$$\forall p \notin P_{\mathcal{A}}: \sum_{(t,\pi(b^*))\in\pi(Y^*) a\in A(p,t)} E(a)\langle\pi(b^*)\rangle \leq (\pi(M_1^*))(p) \qquad (**)$$

Consider annotated places and annotated arcs. From Def. 1 and $(*)$, it follows that:

$$\forall p \in P_{\mathcal{A}}: \pi\Big(\sum_{(t,b^*)\in Y^* a\in A(p,t)} E^*(a)\langle b^*\rangle\Big) \leq \pi(M_1^*(p))$$

which by Defs. 1 and 5.i is equivalent to:

$$\forall p \in P_{\mathcal{A}}: \sum_{(t,b^*)\in Y^* a\in A(p,t)} \pi(E^*(a)\langle b^*\rangle) \leq (\pi(M_1^*))(p)$$

which by $(\dagger)$ and the definitions of $\pi(b^*)$ and $\pi(Y^*)$ is equivalent to:

$$\forall p \in P_{\mathcal{A}}: \sum_{(t,\pi(b^*))\in\pi(Y^*) a\in A(p,t)} E(a)\langle\pi(b^*)\rangle \leq (\pi(M_1^*))(p)$$

which together with $(**)$ and the enabling rule gives us that $\pi(M_1^*)[\pi(Y^*)\rangle$.

Next we have to prove that the marking reached when Y^* occurs in M_1^* covers the marking that is reached when $\pi(Y^*)$ occurs in $\pi(M_1^*)$, i.e. that $\pi(M_1^*)[\pi(Y^*)\rangle\pi(M_2^*)$. A proof similar to the above can be used to show this, and the proof is therefore omitted.

Property iv. We must show that $\forall\ M_1,M_2 \in \mathbb{M},\ \forall\ Y \in \mathbb{Y},\ \forall\ M_1^*,M_2^* \in \mathbb{M}^*$:

$$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$$

Let $M_1[Y\rangle M_2$ in CPN be given. It is straightforward to show that it is always possible to find $M_1^* \in \mathbb{M}^*$ such that $\pi(M_1^*)=M_1$, thus the proof is omitted. Since CPN* is a matching CP-net that is derived from an annotated CP-net with a

sound annotation layer, and $\pi(M_1^*)=M_1$, Def. 6 tells us that there exists $Y^* \in \mathbb{Y}^*$ such that $\pi(Y^*)=Y$ and $M_1^*[Y^*\rangle$.

We have only left to show that the marking reached after Y occurs in M_1 is covered by the marking reached when Y^* occurs in M^*. Since $M_1[Y\rangle M_2$ in CPN, the *occurrence rule* (Def. 2.9 in [70]) gives us that:

$$\forall\, p \in P\text{: } M_2(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle \qquad (\diamond)$$

Since $M_1^*[Y^*\rangle$ in CPN^*, the occurrence rule gives us that:

$$\forall p \in P^*\text{: } M_2^*(p) = \Big(M_1^*(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E^*(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E^*(a)\langle b^*\rangle \quad (\diamond\diamond)$$

In other words, $M_1^*[Y^*\rangle M_2^*$. We must now show that $\pi(M_2^*)=M_2$.

We will show that $\pi(M_2^*)=M_2$ for non-annotated places. We have found M_1^*, such that $\pi(M_1^*)=M_1$. We have that $M_1^*=\pi(M_1^*)$ and $M_2^*=\pi(M_2^*)$ for non-annotated places (by Def. 5.i). For all non-annotated arcs $E^*=E$ (by Def. 4.viii). It follows from these facts and $(\diamond\diamond)$ that:

$$\forall p \notin P_{\mathcal{A}}\text{: } (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E(a)\langle b^*\rangle$$

which by the fact that $\pi(b^*)=b^*$ for all variables in Var(E(a)) (by Def. 5.ii) and the fact that $\pi(Y^*)=Y$ is equivalent to:

$$\forall p \notin P_{\mathcal{A}}\text{: } (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle \quad (\diamond\diamond\diamond)$$

We will show that $\pi(M_2^*)=M_2$ for annotated places. From the definition of π for multi-sets and markings (Defs. 1 and 5.i) and from $(\diamond\diamond)$, it follows that:

$$\begin{aligned}\forall p \in P_{\mathcal{A}}\text{: } (\pi(M_2^*))(p) = {} & \Big((\pi(M_1^*))(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} \pi\big(E^*(a)\langle b^*\rangle\big)\Big) \\ & + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} \pi\big(E^*(a)\langle b^*\rangle\big)\end{aligned}$$

which by $(\dagger)$ and the fact that $\pi(M_1^*)=M_1$ is equivalent to:

$$\forall p \in P_{\mathcal{A}}\text{: } (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E(a)\langle b^*\rangle$$

which by the definitions of $\pi(b^*)$ and $\pi(Y^*)$, and the fact that all variables in E(a) are bound by $\pi(b^*)$ is equivalent to:

$$\forall p \in P_{\mathcal{A}}\text{: } (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle$$

which together with $(\diamond)$ and $(\diamond\diamond\diamond)$ gives us that $\pi(M_2^*)=M_2$.

□

Chapter 10

Case Study: Analysis of Web Servers

The paper "Simulation Based Analysis of Web Servers" presented in this chapter has been published as a workshop paper [127].

[127] L. Wells, S. Christensen, L. M. Kristensen, and K. Mortensen. Simulation based performance analysis of web servers. In R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 59–68. IEEE, 2001.

This chapter is, except for minor typographical changes, the same as the paper [127].

Simulation Based Performance Analysis of Web Servers

Lisa Wells* Søren Christensen* Lars M. Kristensen*
Kjeld H. Mortensen*

Abstract

This paper presents a general framework for modeling distributed computing environments for performance analysis by means of Timed Hierarchical Coloured Petri Nets. The proposed framework was used to build and analyze a Coloured Petri Net model of a HTTP web server. Analysis of the performance of the web server model reveals how the web server will respond to changes in the arrival rate of requests, and alternative configurations of the web server model are examined. These are the results of a research project conducted in cooperation between the CPN Centre and Hewlett-Packard Corporation on capacity planning and performance analysis of distributed computing environments.

10.1 Introduction

The Internet and the World Wide Web (WWW) have experienced exponential growth since their inception. Popular web sites receives millions of hits per day, and it is not uncommon for these sites to exhibit extremely high response times. High response times are a source of frustration for users, and with the growing use of web sites for e.g., electronic commerce this may damage the reputation of the company offering the web site, leading to loss of business. As a consequence, it is important to be able to identify bottlenecks, predict future capacity shortcomings, and determine the most adequate or cost effective way to reconfigure such distributed computing environments to overcome performance problems and cope with increasing workload demands.

This paper presents one of the first results of the CPN Centre which is a research project conducted as a cooperation between the CPN Group at the University of Aarhus and Hewlett-Packard (HP) Corporation. One of the goals of the CPN Centre is to investigate the use of Coloured Petri Nets (CP-nets or CPNs) [70, 71, 72, 80] and simulation as an underlying technology for performance analysis and capacity planning of distributed computing environments. In the initial project phase the goal has been to establish a proof-of-concept. It was therefore decided to consider a concrete representative example of a

*CPN Centre, Department of Computer Science, University of Aarhus, Åbogade 34, 8200 Århus N, Denmark. E-mail: wells,schristensen,lmkristensen,khm@daimi.au.dk.

distributed computing environment in the form of a simple web server environment. It is the main results of this first subproject which is the subject of this paper.

The main result of the first subproject has been the development of a modeling framework for distributed computing environments. This framework is based on a building-block approach dividing the components of the CPN models into three distinct layers: a structural layer describing clients, servers, networks, and their relationship; an application layer describing the applications running on the servers and the clients; and a resource layer describing the resources, i.e., CPUs, disks, and communication channels, of the system. This means that the CPN models includes both a functional view of the system represented by the application layer, and a performance view represented by the resource layer.

A second result has been the simulation based performance analysis of the constructed CPN model. The CPN model was validated and calibrated by comparing performance results from simulation with the performance which can be observed in a corresponding physical environment, for details see [79]. Using the validated model, simulation experiments were run in order to examine the effects of varying the arrival rates of requests on the performance of the web server. Additional experiments were undertaken in order to compare the performance of different configurations, e.g. faster CPU and faster disk, with the basic configuration corresponding to the actual web server.

For construction and simulation of the CPN models we rely on the Design/CPN tool [31, 40]. The simulator of Design/CPN has previously been used in other projects on performance analysis e.g., in the areas of high-speed interconnects [32], and ATM networks [36]. For collecting data about the performance of the CPN web model during simulations, we have used the Design/CPN Performance Tool [87]. This tool makes it possible to collect data about the performance of the considered system during lengthy simulations without having to modify or make extensions to the CPN model itself.

In the literature, there are several papers on performance analysis of web servers. Many of these papers present measurement studies that focus on workload characterization [6, 14, 82] or measurement of, e.g., resource utilization and response time [2, 41, 68]. Analytic models have been used to analyze the performance of HTTP over several transport protocols [61] and for capacity planning of web servers [41]. As one of the few simulation studies of web servers, [123] presents an end-to-end queueing model of a web server environment. Although these studies provide excellent insight into the performance of web servers, and have made significant contributions to the understanding of web workloads, none provide a framework that can be used for both performance analysis and functional analysis of web servers, in particular, and of distributed systems, in general. Furthermore, the proposed framework can be useful for answering "what if" questions concerning possible alternative web server configurations or workloads.

This paper is organized as follows. Section 10.2 provides the necessary background on web servers and timed hierarchical CP-nets for understanding the rest of the paper. The reader is assumed to have some background knowledge on the basic concepts of High-level Petri Nets [73]. Section 10.3 describes the

physical web server, and explains how workload for the web server and the simulation model was obtained. Section 10.4 introduces the framework for modeling of distributed computing environments and illustrates how it can be used to model a HTTP web server. Section 10.5 presents performance results obtained by making simulations of the constructed CPN model. Finally, Sect. 10.6 sums up the conclusions from the project.

10.2 Background

This section provides the background on web servers, and the time and hierarchy concepts of CP-nets sufficient for understanding the rest of this paper. Section 10.2.1 reviews web servers and the HTTP protocol. A detailed treatment of these concepts can be found in e.g., [115]. Section 10.2.2 reviews the time concept of CP-nets. Section 10.2.3 reviews the hierarchy concepts of CP-nets. A detailed treatment of the time concept and the hierarchy concept of CP-nets can be found in [71] and [72], respectively.

10.2.1 Web servers

The fundamental service offered by a web server is access to the documents stored at the web server to *web clients*. These documents are typically Hypertext Markup Language (HTML) [97] documents but may also be e.g., graphics or plain ASCII files. Documents stored at the web server are typically accessed by web clients using a web browser. The main components of a web server consist of a hardware platform (e.g., CPU and disk), an operating system, the web server application, and the documents stored at the web server.

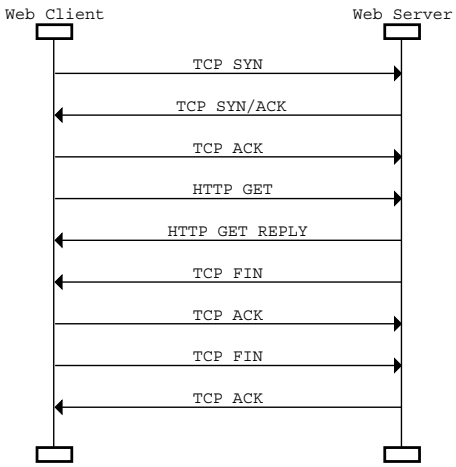


Figure 10.1: HTTP Client-Server Communication.

Communication between the server and the clients is based on the Hypertext Transfer Protocol (HTTP) [115]. HTTP is an application level protocol build on top of TCP/IP [115]. Communication between a client and a server

is always in the form of a request-response pair initiated by the client. The Message Sequence Chart [67] shown in Fig. 10.1 depicts the typical communication between a server (Web Server) and a client (Web Client) which successfully requests and receives a document from the server. The topmost three arrows corresponds to the opening of a TCP connection between the client and the server. The opening of a TCP connection consist of a three-way handshake (exchange of the messages TCP SYN, TCP SYN/ACK, and TCP ACK) between the client and the server. When the TCP connection has been established, the client requests the document by sending a HTTP GET message to the server specifying the document requested. In the following this will be referred to as a *get-request*. The server will now retrieve the document from its disk or possibly from memory if cached. Caching can improve web server performance, and the caching strategy can be chosen to maximize cache hit rate or minimize disk I/O's dependent on the bottleneck of the web server in question [6]. After retrieving the document, it is then transmitted to the client in one or more HTTP GET REPLY messages using the established TCP connection. Finally, the bidirectional TCP connection is closed in both directions using the TCP FIN and TCP ACK messages.

Throughout this paper we will consider HTTP/1.0 [19]. This means that if a requested HTML document contains e.g., inline images then such images are requested separately using the same communication pattern as described above. In particular this means that a TCP connection is opened for each subdocument. A newer version of the HTTP protocol, i.e. HTTP/1.1 [44], includes the notion of persistent connections to avoid the overhead of opening separate TCP connections. It is rather straightforward to modify the CPN model presented in this paper to reflect HTTP/1.1.

10.2.2 Timed Coloured Petri nets

The time concept of CP-nets is based on the introduction of a *global clock*. The clock values represent *model time*, and they may either be integers, i.e. discrete time, or reals, i.e. continuous time. In addition to the token value, we allow each token to carry a time value, also called a *time stamp*. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e., removed by the occurrence of a binding element.

In a timed CP-net a binding element is said to be *colour enabled* when it satisfies the enabling rule for untimed CP-nets, i.e., when the required tokens are present at the input places and when the guard of the transition evaluates to true. However, to be enabled, the binding element must also be *ready*. This means that the time stamps of the tokens to be removed must be less than or equal to the current model time.

The marking of places with a *timed colour set (type)* is a *timed multi-set* of token values. A timed multi-set is similar to a multi-set, except that each member of the multi-set carries a time stamp. To model that an event takes r time units, we let the corresponding transition create time stamps for its output tokens that are r time units larger than the clock value at which the transition occurs. This implies that the tokens produced are unavailable for r time units.

The execution of a timed CP-net is time driven, and it works in a way similar to that of event queues found in many 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).

10.2.3 Hierarchical Coloured Petri nets

The basic idea underlying hierarchical CP-nets is to allow the modeler to construct a large model from a number of smaller CP-nets called *pages*. These pages are then related to each other in a well-defined way as explained below.

In a hierarchical CP-net, it is possible to relate a so-called *substitution transition* (and its surrounding places) to a separate CP-net called a *subpage*. A subpage provides a more precise and detailed description of the activity represented by the transition. Each subpage has a number of *port places* and they constitute the interface through which the subpage communicates with its surroundings. To specify the relationship between a substitution transition and its subpage, we must describe how the port places of the subpage are related to so-called *socket places* of the substitution transition. This is achieved by providing a *port assignment*. When a port place is assigned to a socket place, the two places become identical. The port place and the socket place are just two different representations of a single conceptual place. More specifically, this means that the port and the socket places always have identical markings.

It should be noted that substitution transitions never become enabled and never occur. Substitution transitions work as a macro mechanism. They allow subpages to be conceptually inserted at the position of the substitution transitions – without doing an explicit insertion in the model.

10.3 The ITCHY environment

In this section we describe the hardware and software constituting the performance analysis and capacity planning laboratory environment. The environment is used for making performance measurements which can be compared with the simulation results of the CPN model. The capacity planning laboratory environment has been named ITCHY after the name of the web server. A detailed description of the ITCHY environment can be found in [79].

10.3.1 Server configuration

The web server is an Intel Pentium II 266 MHz processor based PC equipped with a local disk, and 160 MB RAM of internal memory. The server is running Windows NT 4.0 and has a Microsoft web server application installed. For the experiments the web server application was configured as a single threaded web server with cache disabled. The aim has been to configure the web server and

choose parameters in a way which makes it possible to put heavy load on the server by means of a single client (to which we will return in a moment) and relatively few requests. As a consequence, the configuration is not realistic and is far from optimal with respect to performance of the web server. However, this rather disabled configuration is better suited for initial experiments and for judging the accuracy of the CPN model. In order to model a multi-threaded server, the only change that needs to be made in the CPN model is to change the value of a parameter which determines the number of web server threads.

10.3.2 Workload model

It is obviously impractical to involve thousands of users to generate a realistic web workload. Thus we need to generate the workload by means of software. In this section we explain how a sufficiently realistic workload can be generated such that performance measurements for different parameter configurations can be obtained.

The approach taken is that of using a trace client. A *trace client* takes as input a web access log file and then makes a replay of the get-requests contained in the web access log file. Each get-request (entry) in a web access log contains a time stamp, specifying when the requests are to be made, and a specification of the document requested. The log can be real or automatically generated. A real log extracted from some existing web server is easy to use with the trace client but is lacking flexibility with respect to the variation of experiments. Automatically generated log files are harder to create because one needs to investigate statistical models, however the flexibility is more beneficial compared with using a real log file.

We made a simple statistical model of workloads with focus on file size distribution, because it is one of the important factors in a realistic workload. In order to estimate the distribution of file sizes, a web access log file covering a period of one month was analyzed. The Windows-based application *Curve Expert* determined that the best fit was the Weibull distribution [84], with the following distribution function: $F(x) = 1 - 1.2393187 * \exp^{-0.024458885x^{0.475}}$

Figure 10.2 shows the fitted Weibull distribution and the file sizes from the web access log file in question (DAIMI log). The average file size is approximately 5KB. To avoid any effect related to the file cache of the web server all get-requests in the generated workload are for distinct files.

Due to time constraints, a detailed analysis of neither the request times nor temporal locality of requests in the web access log file was undertaken. The log file covered a period of one month, and it was observed that the average request rate was 1 request/second. In the following, we will assume that the workload during peak hours is 5-10 requests/second, which is not an unreasonable expectation judging by what has been observed within other academic environments [6]. Furthermore, we will assume that the interarrival times between requests are exponentially distributed. This is not the most accurate model for request arrivals [6], but it proved to be sufficient for our purposes. In future experiments, the *SURGE* [14] tool could be used to generate accurate workload for web servers.

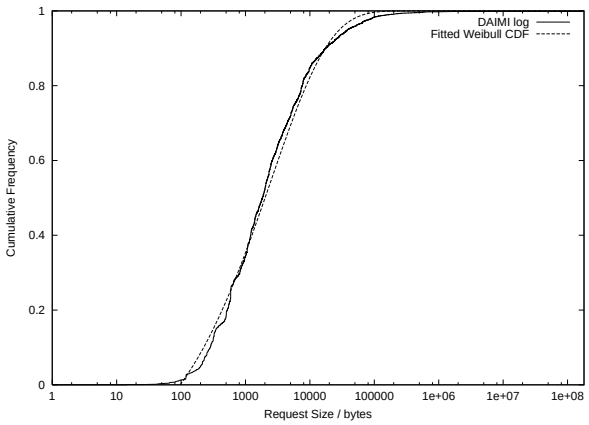


Figure 10.2: Distribution of file sizes.

10.4 Modeling framework

In this section we present the developed modeling framework. As we introduce the basic ideas in the modeling framework, we will also present the constructed CPN model of the web server. In the following, the CPN model of the web server will be referred to as the *CPN web model*.

The modeling framework is based on the idea of logically dividing the pages of the CPN model into three layers: the *structural layer* representing the system architecture; the *application layer* representing the various applications running on the clients and servers of the modeled system; the *resource layer* representing the resources present in the considered system.

The structural layer of the modeling framework is motivated by the observation that a distributed computing environment, by definition, consists of several interconnected computers or workstations communicating using, e.g., local area networks (LANs), wide area networks (WANs), routers and gateways. Therefore, the model has to identify these components as well as their relationship which constitutes the architecture of the distributed computing environment under consideration.

The application layer of the modeling framework is motivated by the observation that a main component in a distributed computing environment is the applications e.g., the web servers, database servers, and file servers running on the different workstations. The relationship between these applications are often based on the client/server paradigm. The communication between clients and servers in the form of requests and responses has a high impact on the performance of a distributed computing environment. It is typically the performance of the service offered by these applications which is of interest when analyzing the performance of a distributed system.

When applications are executing and communicating, they make use of the resources in the environment e.g., CPUs, disks, LANs, and WAN connections. It is the use of these resources imposed by the different applications which determines the utilization of resources and the performance of the considered

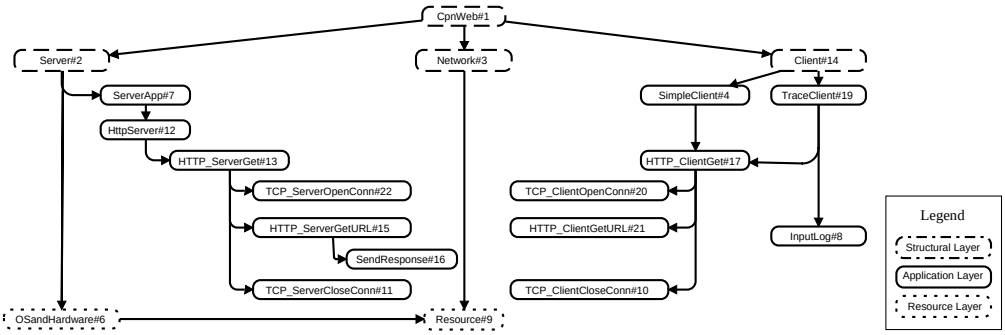


Figure 10.3: CPN web model overview.

system. Since we aim at conducting performance analysis of distributed computing environment, these resources must also be represented in the models.

Although the idea of using a layered approach is not new in the area of distributed systems, e.g. consider the OSI layers for network architectures [115], we have not seen another *modeling framework* that explicitly divides the functional view and the performance view of a system into separate layers within a model. Separating the parts of the model giving a functional description (the application layer) from the part of the model describing the use of resources implies that it is easy to make the transition from a CPN model focusing on performance to a CPN model focusing on the logical correctness of the system. This can be done by simply disabling the pages in the resource layer since all aspects of the CPN model related to performance are isolated in this layer. The separation also means that during the development of a CPN model, one can first construct the structural and application layers of the CPN model, validate that this functional description behaves as expected, and subsequently add the use of resources and conduct performance analysis of the considered system. Of course, one must ensure that the resource layer does not affect the functionality of the application and structural layers. Unfortunately, we were unable to do any functional analysis of our model, due to time constraints. The focus of this particular project was to investigate the use of Coloured Petri Nets and simulation for performance analysis. However, in the future, we do intend to use our modeling framework to do both functional and performance analysis of distributed computing environments.

Figure 10.3 provides an overview of the pages constituting the CPN web model. Each node in Fig. 10.3 represents a page in the CPN model, and is named according to the page in the CPN model it represents. An arc leading from one node to another node means that the latter is a subpage of the former. The pages CpnWeb, Server, Network, and Client are the pages at the structural layer of the CPN model. They describe the architecture of the considered system as consisting of a server part (left), a client part (right), and a network part (middle).

The pages Resource and OSandHardware constitute the resource layer of the CPN model. In the CPN web model there are three different resources: the

CPU resource of the server, the disk resource of the server, and the network resource corresponding to the LAN between the server and the client. We do not consider, e.g., the CPU resource of the client part since our main concern will be to analyze the performance of the server and not the performance of the clients. The OSandHardware page is used only to initialize the server resources, and the Resource page is described in more detail below.

The remaining pages are associated with the application layer of the CPN model, and they constitute four basic building blocks: a HTTP web server building block, a web client building block, a TCP protocol building block, and a building block describing parts of the HTTP protocol. The TraceClient page generates requests according to a web access log file, such as those described in Sect. 10.3. Page InputLog (lower right) enables the CPN model to read in web access log files and to use these as workload specification for driving the simulations. The SimpleClient page generates requests on-the-fly based on a user-defined workload specification.

In this paper we will not go into detail with all pages of the CPN web model. In the following three subsections we will instead present selected examples of pages from the three different layers of the CPN model.

10.4.1 Structural layer

We now give an example of a page from the structural layer of the CPN web model. Figure 10.4 depicts the CpnWeb page which provides the most abstract view of the model. The substitution transitions Server, Network, and Client correspond to the the three main parts of the CPN model. Place Network Buffer is used to model packets in transit between the clients and the server.

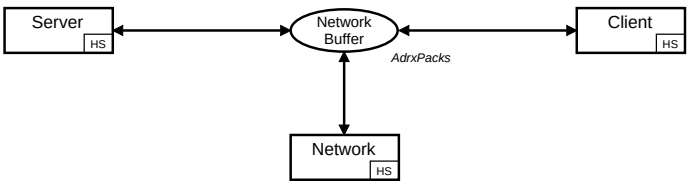


Figure 10.4: The CpnWeb Page.

10.4.2 Application layer

The application layer of the CPN web model is made up of the HTTP web server, which is the only application under consideration in the presented model. Below we give two examples of pages from the application layer of the server part of the CPN web model.

Figure 10.5 depicts the page HTTP_ServerGet. This page models the program executed by the threads of the HTTP server. Each thread repeatedly executes a loop in which get-requests from the clients are processed. The loop consists of the opening a TCP connection, represented by the substitution transition

TCP_OpenConn, handling the request (HTTP_GetURL), and closing the TCP connection (TCP_CloseConn). All three phases require the execution of certain jobs which are put as tokens on place Jobs. We will return to how jobs are represented when we present the resource layer of the CPN web model. All three phases require access to the network represented by place Server Buffer, and handling a get-request may require access to the memory cache of the HTTP web server, represented by place Cache.

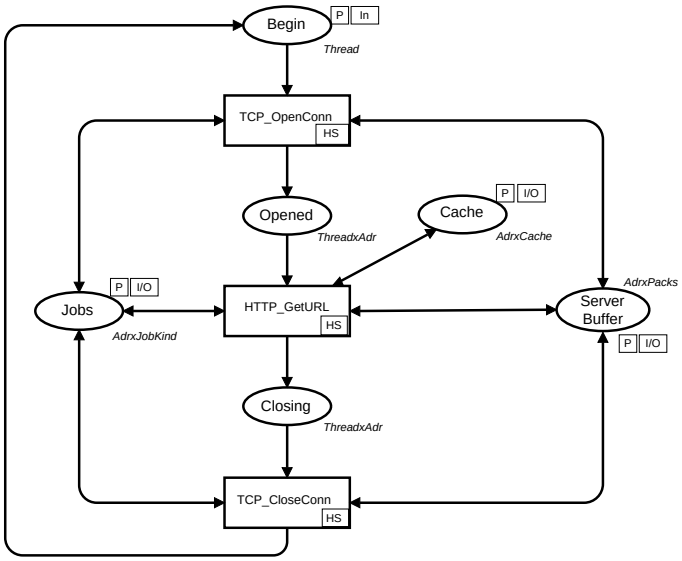


Figure 10.5: The HTTP_ServerGet Page.

To illustrate the interplay between the application layer and the resource layer in terms of how the application layer requests resources at the resource layer, we now consider the page SendResponse which describes the behavior of the HTTP web server when sending the requested document back to the client. This page is shown in Fig. 10.6. Note that SendResponse is a subpage of HTTP_ServerGetURL (not shown) which is, in turn, a subpage of HTTP_ServerGet.

Sending the response of the get-request back to the client consists of the two phases modeled by the transitions Read URL and Send Response. Transition Read URL models the web server thread, represented by (i,app), checking the file attributes of the requested document, fetching the document from either cache or disk, writing the get-request into the servers web access log, and possibly updating the cache. Jobs for the CPU and disk resources are created by the function CreateJobs on the arc from Read URL to Jobs. Disk resources are always needed to read the file attributes of the document and to write the server access log. If the response code is HTTP_OK, then additional resources are needed. CPU resource is always needed to search through the cache, and in case of a cache hit, CPU resource is needed to read the document from main memory. In case the document is not cached, some disk resource is needed to read the document from the disk. When these jobs have been executed, as described by the arc inscription on the arc from the place Jobs to the transition Send Response,

the response can be sent. Transition Send Response puts a packet on the network according to the determined response.

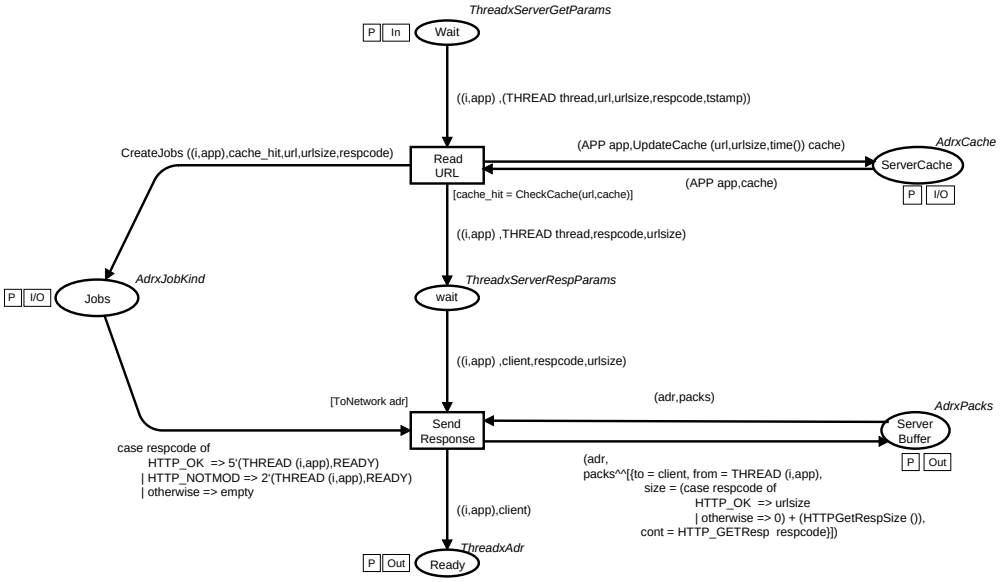


Figure 10.6: The SendResponse Page.

10.4.3 Resource layer

A *resource* is characterized by the capability of executing *jobs*. Execution of jobs at resources takes time and determines the progress of time in the CPN model. The concepts of resources and jobs of the resource layer facilitate making performance analysis based on the CPN model. In this modeling framework, the resource layer of the CPN model is the only part of the CPN model where the time constructs of CP-nets are directly exploited. They are only exploited indirectly at the application layer using the interface between the application and the resource layer as illustrated in Fig. 10.6.

The page Resource, shown in Fig. 10.7, is the central page at the resource layer of the CPN web model and will be explained in more detail below. Recall that three kinds of resources are currently considered: the *CPU resource* of the server, the *disk resource* corresponding to the local disk of the server, and the *network resource* between the clients and the server. In Fig. 10.7 the resources are located as tokens on the place Resource. Resources are described by the colour set Resource which identifies the different attributes of a resource, e.g. the state attribute reflects whether the resource is idle or running, and the parameter attribute describes, among other things, the speed of the resource.

Now let us return to Fig. 10.7. Jobs arrive for the resource at the input/output port place Jobs. The different kinds of jobs are described by the colour set JobKind. Each kind of job has an attribute which specifies the size of the job. When a thread or an application requests use of a resource, e.g., a CPU resource, it will put a token on place Jobs, as was illustrated in Fig. 10.6.

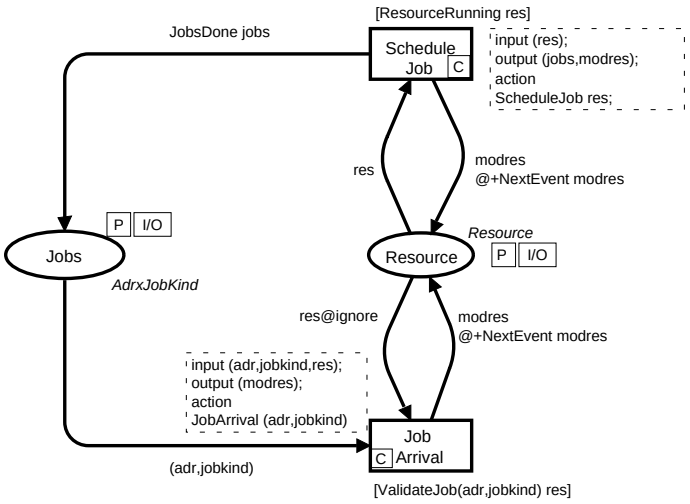


Figure 10.7: The Resource Page.

When the job has been executed at the resource, a token will be put on place Jobs to signal that the job has been processed. For example, when a CPU or disk job has been executed, it becomes *READY*. When a network job has been executed, i.e., the packet has been transmitted, it becomes *TRANSMITTED*.

Transition Job Arrival adds a newly arrived job to the job queue of the corresponding resource. Function `ValidateJob` ensures that the job and the resource match. This makes sure, for example, that threads can only request CPU resources of the computer on which the thread is executing. Transition Schedule Job is responsible both for scheduling jobs at the resource and for removing completed jobs.

The colour set Resource of place Resource is timed. The time stamp of a resource token residing on this place is calculated by the function NextEvent, and it corresponds to the time at which the next scheduling of a job on the resource is to take place. This is a very important construction in order to make simulations tractable since it, in practice, significantly reduces the number of steps which have to be executed during simulations.

10.4.4 Model validation

We will now consider both the validation of the logical behavior of the model and the calibration of the parameters of the model. A detailed description of the validation and calibration processes can be found in [79, 95].

The size and complexity of the CPN web model precluded using state space analysis to fully validate the functional correctness of the model, but interactive simulations and Message Sequence Charts were used extensively to validate the logical behavior of the model. While these techniques can never ensure that a model is error-free, they are useful for finding obvious modeling errors, and they can confirm that the model behaves as expected in a variety of situations.

The goal of the calibration process is to obtain a match between the perfor-

mance results obtained by simulation and the performance results measured in the ITCHY environment and in this way to obtain a validated CPN web model. In this phase web access log files were used to generate the same requests for both the ITCHY environment and the CPN web model. The CPN web model has a number of parameters which can be adjusted. The values for many of these parameters were known, e.g. size of TCP packets, the speed of the server disk, and the effective speed of the network. However, the main difficulty was estimating the parameters of the CPU and the parameters related to the use of CPU resource, e.g., establishing a TCP connection or processing a get-request. The process of calibrating the CPU parameters consisted of a number of iterations of adjusting parameters in the model, running simulations, and comparing the results to the performance of the actual web server. For a more thorough description of the calibration process, see [95].

10.5 Performance analysis

In this section we show how the performance of the web server can be analyzed by means of the CPN web model presented in the previous section. We do so by illustrating the kind of performance results which can be obtained from lengthy simulations of the CPN web model.

10.5.1 Simulation experiments

This section briefly describes the simulation experiments that were run. During the performance analysis phase of the project, workload for the CPN web model was generated on-the-fly. The requests that were dynamically generated during a simulation fulfilled the same criteria as the workload that was described in Sect. 10.3.2, i.e., the file sizes of the requested documents were generated from the Weibull distribution, and the interarrival periods between requests were exponentially distributed. The performance of the web server was examined for request arrival rates of 5, 10, 20, 40 and 50 requests/second. Ten independent simulations were executed for each system configuration that was examined, and each simulation corresponded to 30 minutes of activity in the web server environment. 95% confidence intervals were calculated for the performance measures in question.

10.5.2 Performance measures

When considering the performance of a web server, several performance measures are of interest. *Resource utilization* of the CPU and local disk in the web server and of the network denotes the percentage of time in which each resource is busy processing jobs. *Response time*, i.e. the time from a client initiates a get-request by opening the TCP connection until the response is received, is also of interest. This is the delay observed from the point-of-view of a client. Response time is highly variable since it depends largely on the size of the document that has been requested.

10.5.3 Changing arrival rates

This section focuses on the performance of a web server under different workload intensities. Recall that we have assumed that requests arrive at a rate of 5-10 requests/second during peak hours. Let us consider how the performance of the web server may be affected by an increase in the arrival rate of requests.

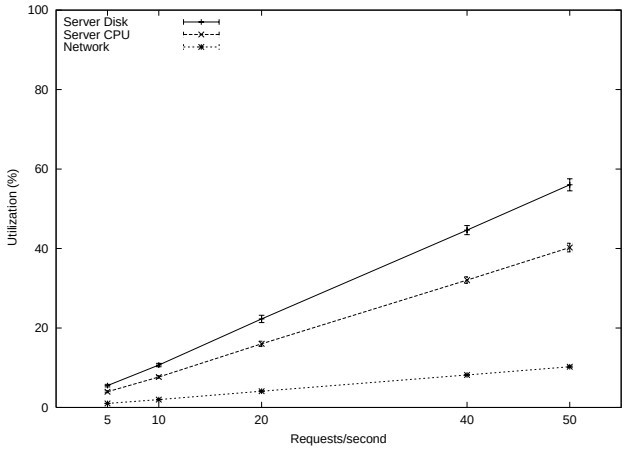


Figure 10.8: Resource utilization.

Figure 10.8 shows the 95% confidence intervals for the average utilization of the different resources under different workload intensities. The disk utilization is largest and the network utilization is lowest for all arrival rates. When the arrival rate is 10 request/second, the average utilization of the server's disk is 10.67%, the average utilization of the CPU of the web server is 7.66%, and the average utilization of the network is 1.98%. Increasing the arrival rate of requests to 50 requests/second, increases the average utilization of the web server's CPU and disk to 40.25% and 56.03%, respectively. The average utilization of the network is also increased to 10.23%. The increase in utilization of the network is not as large as for the resources in the web server since the capacity of the network is quite large (5 Mb/sec.).

The average response time is also affected by the arrival rate of requests to the server. Figure 10.9 shows the 95% confidence intervals for average response time for the aforementioned arrival rates. When the request arrival rate is 10 requests/second, then the average response time is 17.37 ms, i.e. the client receives a response almost immediately. For a much heavier workload of 50 requests/second, the average response time is 82.44 ms. Even though the response is still very fast, it is noticeably slower for heavy workloads.

10.5.4 Alternative configurations

The CPN web model can also be used to analyze and compare the performance of alternative configurations of the web server. Suppose that the peak arrival rate of requests doubles to approximately 20 requests/second. The utilization of the resources of the server will increase, the quality of service that the web

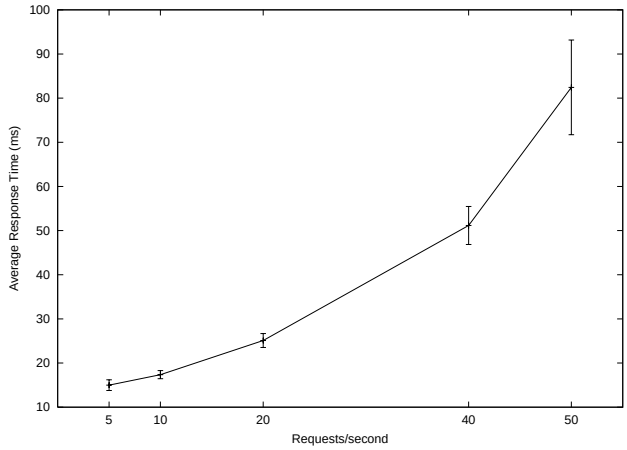


Figure 10.9: Average response time.

server can provide will be affected, and response times for clients may increase. Let us examine several alternative configurations of the web server. Suppose that the available options are:

1. The configuration of the CPN web model corresponding to the ITCHY web server.

2. As in 1, but with a CPU that is twice as fast.

3. As in 1, but with a disk that is twice as fast.

4. As in 1, but using a cache that can hold a large percentage of the most frequently requested documents.

Ten independent simulations were run for each of these configurations, and several comparisons of the different configurations were made. *Paired-t confidence intervals* [84] were used to determine whether or not there is a significant difference in the utilization of both the CPU and disk for the configurations that were compared.

Table 10.1 provides an overview of the comparisons that were made and the conclusions that can be drawn from the comparisons. The first column in Table 10.1 is simply a name of the comparison. The numbers in columns 2-4 correspond to the numbered configurations above. Consider, for example, the second row in Table 10.1. The name of the comparison that has been made is *fastdisk*. In this comparison, configurations 3 and 1 were compared. The paired-t confidence intervals for this comparision reveal that the disk utilization for configuration 3 is lower than the disk utilization for configuration 1, as is expected. Based on the observations that have been made, there is no significant difference between the utilization of the CPU for the two configurations.

Comparison	Configuration		Configuration with lowest utilization	
	A	B	CPU	Disk
fastcpu	2	1	2	–
fastdisk	3	1	–	3
cache	4	1	4	4
cpudisk	2	3	2	3
cpucache	2	4	2	4
diskcache	3	4	4	4

Table 10.1: Comparison of configurations.

10.6 Conclusions

In this paper we have presented a framework for construction of CPN models for performance analysis of distributed computing environments. It was demonstrated how the framework can be used to build a CPN model of a web server environment consisting of a HTTP web server and web clients connected to a LAN. In this modeling framework there is a clear separation within a model between the components which model the functionality of the system and the components which model the performance related aspects of the system. Thus, it is relatively simple to observe the functionality by simply choosing to disable the performance related components in the system. Carefully designed experiments and the use of sound statistical methods allow us to make informed choices when we need to improve the overall performance of the system. The model can also be used to investigate the effect on performance of reconfiguring the web server in different ways.

It may be argued that the CPN web model is simplistic in the sense that it only considers a web server with clients connected to the same local area network as the web server. In practice, clients often communicate with a web server using the Internet. However, a main goal of the project was to compare the performance results obtained by simulations with the performance monitored in a corresponding physical environment. Therefore, a web server environment which could be setup and controlled within the scope of the project was important. Along these lines one may instead view the CPN model as modeling a web server for an intranet of a company.

Many simplifying assumptions have been made about the web server and its workload during this project. Future work with the web server model could include: using more realistic workload with proper arrival rates and considering temporal locality in the request stream, using a multi-threaded server, and modeling HTTP/1.1. Recent work investigates the impact of different designs of the TCP protocols [61] and caching strategies [6] on the performance of web servers. This is an aspect of web server performance which we have not considered in this paper. However, it is worth mentioning that with the proposed framework for construction of CPN models, which makes a clear distinction between system architecture, applications, and resources, could also serve as a foundation for investigating different web server design alternatives.

Bibliography

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley & Sons, 1995.

[2] V. Almeida, J. Almeida, and C. Murta. Performance analysis of a WWW server. Technical Report 1996-018, Computer Science Department, Boston University and UFMG, Aug. 1996. http://www.cs.bu.edu/techreports/1996-018-www-performance-analysis.ps.Z.

[3] S. Andradóttir. Simulation optimization. In Banks [12], pages 307–333.

[4] B. Appleton. Patterns and software: Essential concepts and terminology. Online: http://www.enteract.com/∼bradapp/docs/patterns-intro.html.

[5] Arena. Online: http://www.arenasimulation.com/.

[6] M. Arlitt and C. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE Transactions on Networking*, 5(5):631–645, 1997.

[7] Artifex.
Online: http://www.artis-software.com/solutions/artifex/index.html.

[8] R. Bagrodia and M. Gerla. A modular and scalable simulation tool for large wireless networks. In *Proceedings of Performance Tools '98*, 1998.

[9] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, October 1998.

[10] G. Balbo. Introduction to Stochastic Petri Nets. In Brinksma et al. [23], pages 84–155.

[11] P. Ballarini, S. Donatelli, and G. Franceschinis. Parametric stochastic well-formed nets and compositional modelling. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 43–62. Springer, 2000.

[12] J. Banks, editor. *Handbook of Simulation*. John Wiley & Sons, Inc., 1998.

[13] J. Banks. Principles of simulation. In *Handbook of Simulation* [12], pages 3–30.

[14] P. Barford and M. E. Crovella. Generating representative web workloads for network and server performance evaluation. In *Performance '98/ACM SIGMETRICS '98*, pages 151–160. Madison WI, 1998.

[15] F. Bause, P. Buchholz, and P. Kemper. QPN-tool for the specification and analysis of hierarchically combined Queueing Petri Nets. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 1995.

[16] M. Beaudouin-Lafon et al. CPN/Tools: A post-WIMP interface for editing and simulating coloured Petri nets. In J.-M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 71–80. Springer, 2001.

[17] M. Beaudouin-Lafon and W. E. Mackay. Reification, polymorphism and reuse: Three principles for designing visual interfaces. In *Proceedings of Advanced Visual Interfaces*, pages 102–109. ACM Press, 2000.

[18] S. Bernardi, S. Donatelli, and A. Horváth. Compositionality in the GreatSPN tool and its application to the modelling of industrial applications. In K. Jensen, editor, *Practical Use of High-level Petri Nets: Workshop Proceedings*, DAIMI PB-547, pages 127–145, 2000.

[19] T. Berners-Lee, R. Fielding, and H. Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Technical Report RFC 1945, Network Working Group, May 1996. http://www.cis.ohio-state.edu/htbin/rfc/rfc1945.html.

[20] G. Berthelot. Transformations and decompositions of nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986*, volume 254 of *Lecture Notes in Computer Science*, pages 359–376. Springer-Verlag, 1987.

[21] H. Bowman, J. W. Bryans, and J. Derrick. Analysis of a multimedia stream using stochastic process algebra. In C. Priami, editor, *Sixth International Workshop on Process Algebras and Performance Modelling*, pages 51–69, 1998.

[22] E. Brinksma and H. Hermanns. Process algebra and markov chains. In Brinksma et al. [23], pages 183–231.

[23] E. Brinksma, H. Hermanns, and J.-P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*. Springer, 2001.

[24] C. Capellmann, S. Christensen, and U. Herzog. Visualising the behaviour of intelligent networks. In T. Margaria, B. Steffen, R. Rückert, and J. Posegga, editors, *Services and Visualization: Toward User-Friendly*

Design, volume 1385 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 1998.

[25] C. Capellmann and 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*, 1994.

[26] CAPLAN project. Online: http://www.daimi.au.dk/CPnets/CAPLAN/.

[27] L. Cherkasova, V. Kotov, and T. Rokicki. On scalable net modeling of OLTP. In *Proceedings of 5th International Workshop on Petri Nets and Performance Models*, pages 270–279. IEEE Computer Society Press, 1993.

[28] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, November 1993.

[29] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical editor and analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1&2):47–68, November 1995. Special issue on Performance Modeling Tools.

[30] S. Christensen and J. B. Jørgensen. Analysis of Bang & Olufsen's BeoLink audio/video system using coloured Petri nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997.

[31] S. Christensen, J. B. Jørgensen, and L. M. Kristensen. Design/CPN - a computer tool for coloured Petri nets. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems - TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Spriner-Verlag, 1997.

[32] G. Ciardo, L. Cherkasova, V. Kotov, and T. Rokicki. Modeling a Scaleable High-Speed Interconnect with Stochastic Petri Nets. In *Proceeding of PNPM'95*, pages 83–93. IEEE Computer Society Press, 1995.

[33] G. Clark et al. The Möbius modeling tool. In R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, 2001.

[34] E. M. Clarke, J. M. Wing, et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[35] H. Clausen and P. R. Jensen. Validation and performance analaysis of network algorithms by coloured Petri nets. In *Proceeding of PNPM'93*, pages 280–289. IEEE Computer Society Press, 1993.

[36] H. Clausen and P. R. Jensen. Analysis of Usage Parameter Control algorithms for ATM networks. In S. Tohmé and A. Casada, editors, *Broadband*

Communications II (C-24), pages 297–310. Elsevier Science Publishers, 1994.

[37] CPN Tools. Online: http://wiki.daimi.au.dk/cpntools/.

[38] J. C. A. de Figueirdo and L. M. Kristensen. Using coloured Petri nets to investigate behavioural and performance issues of TCP protocols. In K. Jensen, editor, *CPN Workshop 1999*, DAIMI PB-541, 1999.

[39] J. Desel and W. Reisig. Place/Transition Petri nets. In Reisig and Rozenberg [108], pages 122–173.

[40] Design/CPN. Online: http://www.daimi.au.dk/designCPN/.

[41] J. Dilley, R. Friedrich, T. Jin, and J. Rolia. Web server performance measurement and modeling techniques. *Performance Evaluation*, 33(1):5–26, 1998.

[42] DSPNexpress. Online: http://ls4-www.informatik.uni-dortmund.de/~Lindemann/software/DSPNexpress/.

[43] Ex*Spect*. Online: http://www.exspect.com/.

[44] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report RFC 2068, Network Working Group, March 1997. http://www.cis.ohio-state.edu/htbin/rfc/rfc2068.html.

[45] D. Flanagan. *Java in a Nutshell*. O'Reilly, 3rd edition, 1999.

[46] J. M. Forneau, L. Kloul, and F. Valois. Performance modelling of hierarchical cellular networks using PEPA. In J. Hillston, editor, *Proceedings of the Seventh Annual Workshop on Process Algebra and Performance Modelling*, pages 139–154. University of Zaragosa Press, 1999.

[47] G. Franceschinis, C. Bertoncello, G. Bruno, G. L. Vaschetti, and A. Pigozzi. SWN models of a contact center: a case study. In R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 39–48. IEEE, 2001.

[48] R. Gaeta and M. A. Marsan. SWN analysis and simulation of large knockout ATM switches. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 326–344. Springer, 1998.

[49] G. Gallasch and L. M. Kristensen. COMMS/CPN: A communication infrastructure for external communication with Design/CPN. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, DAIMI PB-554, pages 75–91. Department of Computer Science, University of Aarhus, Denmark, 2001.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[51] R. German, C. Kelling, A. Zimmerman, and G. Hommel. TimeNET: a toolkit for evaluating non-markovian stochastic Petri nets. *Performance Evaluation*, 24:69–87, 1995.

[52] S. Gilmore and J. Hillston. The PEPA Workbench: A tool to support a process albegra-based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1994.

[53] gnuplot. Online: http://www.gnuplot.info/.

[54] D. Goldsman and B. L. Nelson. Comparing systems via simulation. In Banks [12], pages 273–306.

[55] S. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri Nets*. PhD thesis, University of South Australia, 2001.

[56] S. Gordon and J. Billington. Analysing the WAP class 2 wireless transaction protocol using coloured Petri nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 207–226, 2000.

[57] GreatSPN. Online: http://www.di.unito.it/∼greatspn/.

[58] S. Haddad. A reduction theory for coloured nets. In *Advances in Petri Nets 1989*, volume 424 of *Lecture Notes in Computer Science*, pages 209–235. Springer-Verlag, 1989.

[59] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[60] B. R. Haverkort. Markovian models for performance and dependability evaluation. In Brinksma et al. [23], pages 38–83.

[61] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE Transactions on Networking*, 5(5):616–630, 1997.

[62] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[63] HiQPN. Online: http://ls4-www.informatik.uni-dortmund.de/QPN/.

[64] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 1979.

[65] HP-CPN Centre. Online: http://www.daimi.au.dk/CPnets/HP/.

[66] INFORMS (Institute for Operations Research and the Management Sciences) Simulation Links and Information. Online: http://www.informs-cs.org/geninfo.html.

[67] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.

[68] A. Iyengar, E. MacNair, and T. Nguyen. An analysis of web server performance. In *GLOBECOM 97*, volume 3, pages 1943–1947. IEEE, 1997.

[69] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991.

[70] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[71] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[72] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[73] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets*. Springer-Verlag, 1991.

[74] C. Kelling. A framework for rare event simulation of stochastic Petri nets using "RESTART". In J. Charnes, D. Morrice, D. Brunner, and J. Swain, editors, *Proceedings of the 1996 Winter Simulation Conference*, pages 317–324, 1996.

[75] W. D. Kelton, R. P. Sadowski, and D. A. Sadowski. *Simulation with Arena*. McGraw-Hill, 1998.

[76] W. D. Kelton, R. P. Sadowski, and D. A. Sadowski. *Simulation with Arena*. McGraw-Hill, 2nd. edition, 2002.

[77] J. P. C. Kleijnen. Experimental design for sensitivity analysis, optimization, and validation of simulation models. In J. Banks, editor, *Handbook of Simulation*. Wiley, New York, 1993.

[78] L. Kleinrock. *Queueing Systems, Vol. 1, Theory*. John Wiley, 1975.

[79] L. M. Kristensen, J. Bogorad, S. Christensen, K. Jensen, B. Lindstrøm, K. H. Mortensen, J. S. Thomasen, and L. M. Wells. *HTTP Web Servers – Part A*. Number HP-CPN-1 in HP-CPN Project Report Series. HP-CPN Centre, Department of Computer Science, University of Aarhus, May 1998. http://www.daimi.au.dk/CPnets/HP/.

[80] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[81] O. Kummer, F. Weinberg, and M. Duvigneau. *Renew – User Guide*. Department for Informatics, University of Hamburg, May 2001.

[82] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web server: Design and performance. *Computer*, 28(11):68–74, Nov. 1995.

[83] C. Lakos. On the abstraction of coloured Petri nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1997.

[84] A. M. Law and W. D. Kelton. *Simulation Modeling & Analysis*. McGraw-Hill, 3rd edition, 2000.

[85] C. Lindemann. DSPNexpress: a software package for the efficient solution of deterministic and stochastic Petri nets. *Performance Evaluation*, 22:3–21, 1995.

[86] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley and Sons, 1998.

[87] B. Lindstrøm and L. Wells. *Design/CPN Performance Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1999. Online: http://www.daimi.au.dk/designCPN/man/.

[88] B. Lindstrøm and L. Wells. Performance Analysis Using Coloured Petri Nets. Master's thesis, University of Aarhus, May 1999.

[89] B. Lindstrøm and L. Wells. Annotating coloured Petri nets. To appear in the proceedings of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'02), 2002.

[90] B. Lindstrøm and L. Wells. Towards a monitoring framework for discrete-event system simulations. To appear in the proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02), 2002.

[91] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling of features and feature interactions in Nokia mobile phones using coloured Petri nets. In J. Esparza and C. Lakos, editors, *Applications and Theory of Petri Nets 2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 299–313. Springer, 2002.

[92] Message Sequence Charts in Design/CPN
Online: http://www.daimi.au.dk/designCPN/libs/mscharts/.

[93] M. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Models*. PhD thesis, University of California at Los Angeles (UCLA), 1981.

[94] G. Moncelet, S. Christensen, H. Demmou, M. Paludetto, and J. Porras. Analysing a mechatronic system with coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):160–167, December 1998.

[95] K. Mortensen, S. Christensen, L. Kristensen, and J. Thomasen. Capacity planning of web servers using timed hierarchical coloured Petri nets. In *Proceedings of HP Openview University Associateion (HP-OVUA'99), 6th Plenary Workshop*, Bologna, Italy, 1999.

[96] K. H. Mortensen. Efficient data-structures and algorithms for a coloured Petri nets simulator. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2001. Available online at http://www.daimi.au.dk/CPnets/workshop01/.

[97] C. Musciano, B. Kennedy, and M. Loukides, editors. *HTML: The Definitive Guide*. O'Reilly & Associates, 3rd edition, September 1998.

[98] W. D. Obal, II and W. H. Sanders. An environment for importance sampling based on stochastic activity networks. In *Symposium on Reliable Distributed Systems*, pages 64–73. IEEE Press, 1994.

[99] Observer pattern. Online: http://ootips.org/observer-pattern.html.

[100] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[101] C. D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction To Simulation Using SIMAN*. McGraw-Hill, 2nd edition, 1995.

[102] PEPA: Performance Evaluation Process Algebra.
Online: http://www.dcs.ed.ac.uk/home/stg/PEPA/.

[103] C. A. Petri. *Kommunikation mit Automaten*. Schriften des IIM nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962.

[104] Petri Nets Tool Database.
Online: http://www.daimi.au.dk/PetriNets/tools/db.html.

[105] Petri Nets World. Online: http://www.daimi.au.dk/PetriNets/.

[106] J. L. Rasmussen and M. Singh. *Mimic/CPN: A Graphic Animation Utility for Design/CPN*. Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.au.dk/designCPN/libs/mimic/.

[107] J. L. Rasmussen and M. Singh. Designing and analysing a security system by means of coloured Petri nets. In J. Billington and W. Reisig, editors, *Proceedings of ICTAPN*, volume 1091 of *Lecture Notes in Computer Science*, pages 400–419. Springer-Verlag, 1996.

[108] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[109] Renew. Online: http://www.renew.de/.

[110] W. H. Sanders. *Construction and Solution of Performability Models Based on Stochastic Activity Networks*. PhD thesis, University of Michigan, 1988.

[111] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. *Dependable Computing for Critical Applications*, 4:214–237, 1991. Springer-Verlag.

[112] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In Brinksma et al. [23], pages 315–343.

[113] W. H. Sanders, W. D. Obal, II, M. A. Qureshi, and F. K. Widjanarko. The UltraSAN modeling environment. *Performance Evaluation*, 24:89–115, 1995.

[114] SIGMETRICS. Online: http://www.sigmetrics.org/.

[115] W. Stallings. *Data & Computer Communications*. Prentice-Hall, 6th edition, 2000.

[116] Standard ML of New Jersey.
Online: http://cm.bell-labs.com/cm/cs/what/smlnj/.

[117] J. J. Swain. Imagine new worlds: Simulation survey. *OR/MS Today*, pages 38–51, February 1999.

[118] H. A. Taha. *Modeling and SIMNET*. Prentice-Hall, 1988.

[119] TimeNET. Online: http://pdv.cs.tu-berlin.de/∼timenet/.

[120] UltraSAN. Online: http://chaos.crhc.uiuc.edu/UltraSAN/UltraSAN.html.

[121] A. Valmari. The state explosion problem. In Reisig and Rozenberg [108], pages 429–528.

[122] W. M. P. van der Aalst et al. Ex*Spect* 6.4: An executable specification tool for hierarchical colored Petri nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 455–464. Springer-Verlag, 2000.

[123] R. D. van der Mei, R. Hariharan, and P. K. Reeser. Web server performance modeling. *Telecommunication Systems*, 16(3-4):316–378, 2001.

[124] H. M. W. Verbeek and W. M. P. van der Aalst. Woflan 2.0: A Petri net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, 2000.

[125] M. Villapol and J. Billington. Modelling and the initial analysis of the Resource Reservation Protocol using coloured Petri nets. In K. Jensen, editor, *Proceedings of the Workshop on Practical Use of High-Level Petri Nets*, DAIMI PB-547, 2000.

[126] L. Wells. Performance analysis using coloured Petri nets. To appear in the proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02), 2002.

[127] L. Wells, S. Christensen, L. M. Kristensen, and K. Mortensen. Simulation based performance analysis of web servers. In R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 59–68. IEEE, 2001.

[128] Winter Simulation Conference. Online: http://www.wintersim.org/.