Facilitating the Practical Use of Coloured Petri Nets

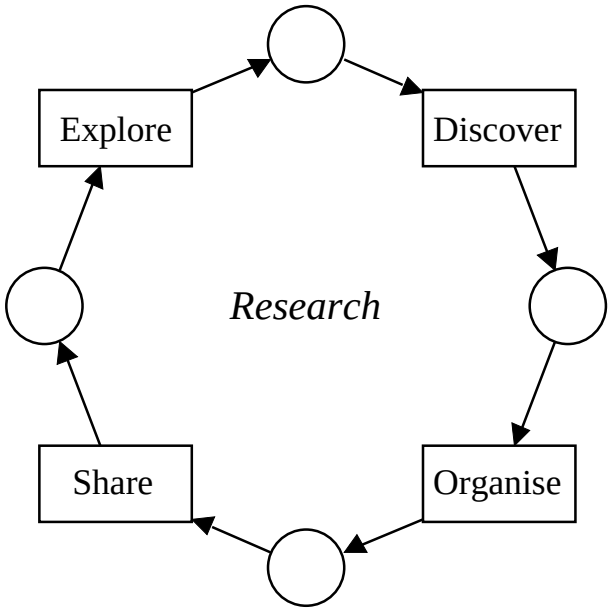
Bo Lindstrøm

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Facilitating the Practical Use of 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
Bo Lindstrøm
30. July 2002

The future throws its shadow
at the present
in the light of the past
– Bo Lindstrøm, 1993.

Preface

Summary

Design and development of systems is a complex task which is even more complex if the systems are distributed and concurrent. Therefore models are often created of such systems to investigate the systems in various ways. Due to the complexity of such models it is essential that modelling tools exist with support for examining the models.

Coloured Petri nets (CP-nets or CPNs) is a formally founded graphical language for modelling and analysis of concurrent and distributed systems. The industrial use of CP-nets has increased during the latest years. This is in particular due to the extensive development of tools supporting various uses of CP-nets.

The primary purpose of this dissertation is to investigate and advance tools for practical use of CP-nets. This includes development of facilities for creating domain-specific graphical user interfaces, a proposal for facilities for including auxiliary information in a CP-net without modifying the CP-net itself, and development of general facilities for extracting information from a CP-net during simulations. The tool facilities have been developed based on concrete needs arising during practical projects, and have when possible been tested in these projects.

The dissertation consists of two parts. Part I is the mandatory overview paper which summarises the research. Part II is composed of five individual papers and constitutes the core of this dissertation. All five papers have been published or accepted for publication as workshop or journal papers.

The overview paper introduces CP-nets, motivates the research presented in this dissertation, and summarises the contents and main contributions of the five individual papers. A substantial part of the overview paper has also been devoted to a discussion of related work for considering the results presented in the papers from a broader perspective.

The first paper describes an approach which allows users without knowledge of CP-nets to control simulations of CPN models and interpret the results obtained from simulations via web-based interfaces. It describes the architecture design of facilities in a simulation tool for making it possible to simulate a CPN model via a web-based interface. The approach is based on giving the modeller the ability to easily create a stand-alone program containing the entire simulator. The initial conditions of the simulator for running several simulations can then be specified via a web page. The fact that the initial conditions of a

simulation can be specified via a domain-specific and user-relevant form, gives users without knowledge of CP-nets and the actual simulation tool the ability to use the pre-constructed simulators for specific analysis purposes. The paper also illustrates that creating so-called batch scripts has some advantages. Batch scripts give the user the ability to run several simulations after having specified initial conditions for each simulation. Thus the user will be able to first specify input via a web page and then based on that input run several simulations.

The second paper discusses CPN models of so-called timed influence nets with logic which is a probabilistic modelling language. Previous work has developed a method to translate a timed influence net into a CP-net. The work in this paper describes a new and more compact translation from timed influence nets with logic into CP-nets. The translation has the property that the net structure of the CP-net will be the same for all translated timed influence nets – only the initial marking changes depending on the actual timed influence net. This more compact translation avoids the generation of simulation code for each timed influence net. The paper also presents some validation results to establish that the CPN models from the two methods are equivalent, i.e. that the new compact CPN model gives the same simulation results as the old less compact model does.

The third paper presents an environment for web-based simulation of influence nets to be used for operational planning. This paper combines the methodologies developed in the first and second paper to conduct temporal evaluation of plans via a web-based graphical user interface for the influence net simulator implemented using CP-nets. Simulating the CP-net in a web environment makes it easy for a subject matter expert to use the CPN simulator for planning without knowing the underlying CPN formalism and tools. This paper discusses the use of influence nets for operational planning, a simulator for influence nets implemented using CP-nets, and the architecture of the complete web-site which can be used for operational planning.

The fourth paper presents a framework for designing and implementing monitoring facilities in simulation tools. Monitoring is any activity related to observing, inspecting, and controlling a simulation of a model. Several libraries and tool extensions have been developed for facilitating visualisation of simulation results, communication with external processes, and performance analysis. Many of these extensions are enabled by integrating ad-hoc code into the CPN model itself. The monitoring framework generalises the monitoring to a uniform and flexible framework which can be used to create various types of monitoring tools without changing the CPN model itself. The framework is presented in general terms that are not specific for CP-nets. The goal of the framework is that it should serve as a reference for implementing different types of monitors in discrete-event system simulators.

The fifth paper discusses how auxiliary information can be added to CP-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 a CP-net. An example of such auxiliary information is packet delay 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. The 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. The set of legal annotations is restricted so that annotations are guaranteed to affect the behaviour of the basic CP-net in a very limited and predictable manner. Annotations are created in annotation layers which makes it easy to disable a set of annotations when using the CP-net for a purpose where the annotations must be disabled.

Acknowledgements

First and foremost, I want to thank Søren Christensen and Kurt Jensen for supervision and inspiration during my Ph.D. studies. Also thanks for hiring me as a student programmer for the Caplan project, and for giving me the opportunity to participate in various workshops, conferences, and summer schools.

Being a member of the Coloured Petri Nets Group (CPN group) at the Department of Computer Science, University of Aarhus, thoughout my Ph.D. study has been very rewarding. In particular because the CPN group consists of people with different views and competences from whom I have learned a lot. They have always been willing to comment on ideas and papers whenever I needed it. I want especially to thank Lisa Wells, Louise Lorentsen, and Thomas Mailund for various discussions and for sharing various opinions. In addition, I am very grateful to Lisa Wells for excellent cooperation on both our common Master's thesis and on common projects during the past three years. I also wish to thank Lars M. Kristensen for help, advise, and inspiration thoughout our common time in the CPN group.

Janice Bogorad and Jan S. Thomasen from Hewlett-Packard also deserve to be thanked for cooperation in the Caplan project and later in projects within the HP-CPN centre.

Thanks to Alexander H. Levis and Lee W. Wagenhals for giving me the opportunity to get insight into the research conducted on military systems by hosting a five months visit to George Mason University (GMU), VA, USA in the spring of 2001. During the visit I also had the pleasure to work with several people whom I am all grateful. Especially, I am very grateful to Lee W. Wagenhals, Sajjad Haider, Insub Shin, and Daesik Kim for good friendship and for various cooperations in the CAESAR project. I also wish to thank Sajjad Haider and Lee W. Wagenhals for participating in writing two of the papers which are part of this dissertation.

Thanks to Ann Eg Mølhave for proof-reading the overview paper contained in this dissertation.

I also wish to thank my parents for raising me to get a natural thirst for knowledge, and for giving me my stubbornness and self-discipline from which I have often benefitted.

Last, but not least, I am deeply grateful for the remarkable patience, support, and encouragement that my girlfriend, Anette, has given me throughout my study.

The work accomplished in this dissertation has been supported by grants from Hewlett-Packard and the Danish National Centre for IT research.

Bo Lindstrøm,
Århus, 30. July 2002

Contents

Part I

Overview

Chapter 1

Introduction

This part of the dissertation provides an overview of the research conducted by the author during the past three years. In addition to the overview part, this dissertation consists of five individual papers.

This chapter is organised as follows. Section 1.1 gives a brief introduction to coloured Petri nets (CP-nets or CPNs). Section 1.2 motivates and gives the aims of the research presented in this dissertation. Section 1.3 gives a brief overview of the research presented in this dissertation, and gives an outline of the remaining chapters in this overview paper.

1.1 Coloured Petri Nets

CP-nets was formulated by Jensen [44, 45, 46] as a formally founded graphically oriented modelling language. CP-nets are useful for specifying, designing, and analysing concurrent systems. In contrast to ordinary Petri nets, CP-nets provide a very compact way of modelling complex systems, which makes CP-nets a powerful language for modelling and analysing industrial-sized systems. This is achieved by combining the strengths of Petri nets with the expressive power of high-level programming languages. Petri nets provides the constructions for specifying synchronisation of concurrent processes, and the programming language provides the constructions for specifying and manipulating data values.

Practical use of CP-nets has been facilitated by developing tools to support construction and analysis of systems by means of CP-nets. The Design/CPN tool [27] is a tool supporting CP-nets which uses a variant of the programming language Standard ML [83] as inscription language, i.e. the programming language used for specifying data types and for manipulating data values.

Figure 1.1 shows an example of a simple CPN model which models a calculator. The purpose of the figure is to give the reader an idea of what a CPN model looks like. The CPN model is divided into two parts. The leftmost part (left of the vertical dashed line) models a cycle of the graphical user interface where a user types in two arguments and an operator on the buttons of the calculator. This is followed by displaying the result of the computation. The rightmost part models the computation based on the numbers and the operator typed in by the user.

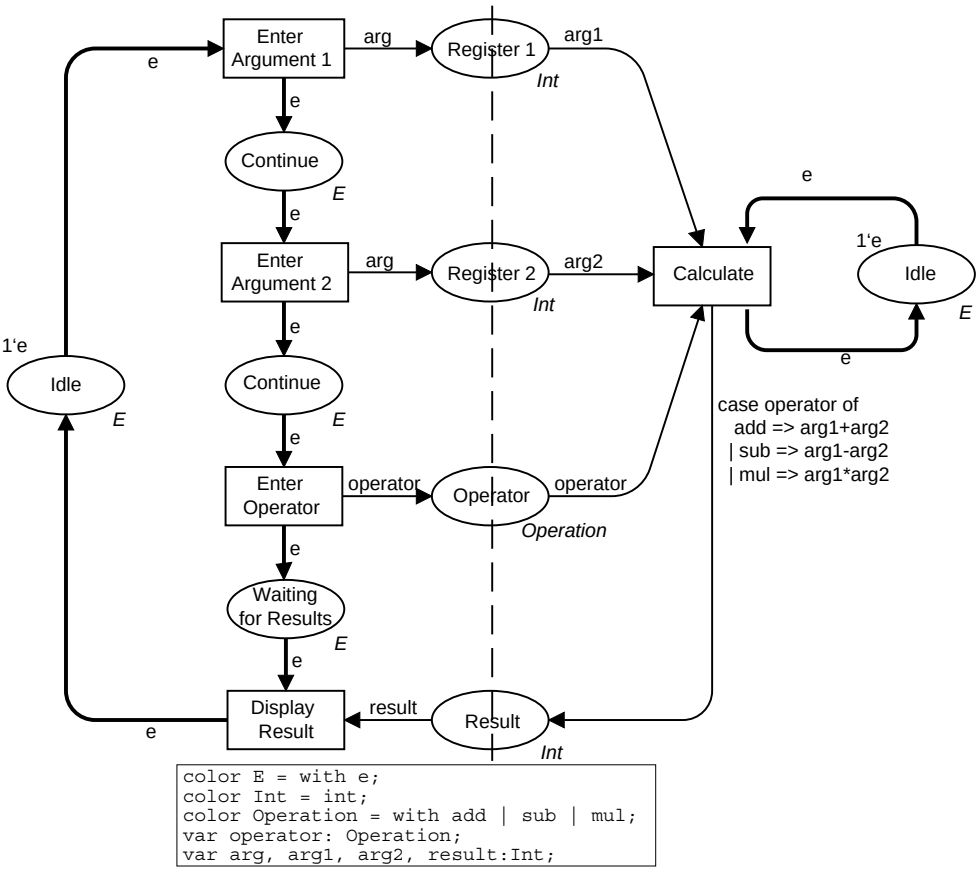


Figure 1.1: CPN model of a simple calculator.

The syntactical elements of a CP-net consist of *places* (drawn as ellipses), *transitions* (drawn as rectangles), *arcs* (connecting places and transitions), and *inscriptions* (text strings associated with places, transitions, and arcs). Places are used to model the state in a system. A state in the context of CP-nets is called a *marking* which represent a distribution of data values (called *tokens*) on the places of the CP-net. Each place has a *colour set* (or type) which specifies the *colours* (or values) of the tokens that the place can hold. The initial distribution of tokens is called the *initial marking*. Transitions are used to model the dynamics or actions in the system. Arcs pointing to a transition are called *input arcs*, while arcs pointing from a transition are called *output arcs*. The arc expressions on input arcs determine what tokens have to be present on *input places* to *enable* the transition. The arc expressions on *output arcs* specify the tokens that will be added to the *output places* when the transition *occurs*. In other words, when a transition is enabled, it may occur and thereby remove tokens from the input places as specified by the expressions on the input arcs, and add the tokens to the output places as specified by the expressions on the output arcs.

A *simulation* is a sequence of consecutive occurrences of transitions, and

corresponds to a single execution of the model. Instead of considering a single simulation of a CP-net, a *state space* can be generated to represent all possible simulations of the CP-net. A state space for a CP-net represents all reachable states and state changes of the system, and can be represented as a directed graph. Each node in the graph represents a reachable marking and each arc represents an occurring *binding element*. A binding element is a pair consisting of a transition and an assignment of data values to the variables appearing in the expressions associated with the transition and the surrounding arcs of the transition. From a state space it is possible to use *state space analysis* to answer a large set of analysis and verification questions concerning the behaviour of the modelled system, e.g. to prove the absence of dead-locks. Given a CP-net, the state space can be generated automatically and analysed in the Design/CPN tool using the Design/CPN State Space Tool [47].

1.2 Motivation and Aims of Dissertation

The work presented in this dissertation has been motivated by experiences with practical use of CP-nets in industrial-sized project. When the work on this dissertation was initiated in 1999, most practical use of CP-nets was conducted by people who had knowledge and experience with CP-nets and the corresponding modelling and analysis tools. However, in particular one industrial project [3] showed that it is sometimes useful to be able to visualise simulations using domain-specific graphics instead of inspecting the token values visually in the CP-net. In other words, the information in the CP-net should be mapped to concepts from the domain of the modelled system. This makes it possible for people without knowledge of CP-nets and the corresponding tools to easily understand and use simulations of CP-nets.

In many industrial projects, CP-nets are used primarily for modelling and evaluating designs. Afterwards, a programming language like e.g. C++ [94] is used for implementing the selected design. However, the project discussed in [3] showed that the modelling language, CP-nets, used for modelling and evaluating the design can also be used as the final implementation language. In the project they extracted the simulator code from Design/CPN and used that simulator code as the final implementation of the application. In other words, CP-nets were not used only in the design phase of the project for creating and analysing a model – they were also used as the implementation language. The main motivations for the work presented in this dissertation comes from using CP-nets as a grahical high-level implementation language and from creating domain-specific graphical user interfaces (GUIs) for CPN simulators.

In order to make practical use of CP-nets accessible to people without knowledge of CP-nets, it was necessary to develop a general method to abstract from the CPN-specific concepts of the interaction with the CPN tools[1]. In other words, facilities were needed for creating domain-specific GUIs for the CPN tools so that it is possible to interact with the CPN tool using domain-specific

[1]The name *CPN tools* refers to any tool used for modelling using CP-nets, and not exclusively to the tool CPN Tools.

concepts instead of CPN-specific concepts. To fulfil these needs, a method had to be developed for creating domain-specific interfaces for simulators of CP-nets. This includes facilities for scripting and customising the control of simulators and analysis tools, and for creating a suitable GUI.

The research during the last years has shown that CP-nets are useful for other purposes than those where CP-nets have their obvious strengths, i.e. modelling and verification of concurrent systems. The need for performance analysis of industrial-sized systems is just one field where the expressive power of CP-nets has proved to be very useful [20, 24, 49, 63]. Performance analysis using CP-nets is often based on data collected during simulations. Another area is data collection for visualisation purposes. Using the programming language Standard ML (SML) which is the inscription language in the Design/CPN tool, it is relatively easy to add SML code to a CPN model for extracting such data. However, when a CPN model is constructed, it is often used for several purposes. The modifications of the CPN model required to conduct a certain analysis, like e.g. performance analysis, may make it difficult to use the CPN model for different purposes. This motivates a need for ways to annotate a CP-net with auxiliary information which can easily be added or removed from a CP-net. In addition, improved facilities for extracting information from a CP-net during simulations are also motivated by this need.

The main objective of the work presented in this dissertation has been to design and develop extensions to CP-nets and facilities for the supporting CPN tools to remedy the shortcomings discussed above.

1.3 Outline of Dissertation

The work presented in this dissertation has been documented in five papers [54, 55, 56, 61, 62]. The papers are contained in Part II in Chaps. 6-9 of this dissertation. Each of these chapters first contain a short description of the publication history and the status of the paper contained in the chapter.

The remaining chapters of Part I are organised as follows:

Chapter 2 summarises three papers which document the research results from the CAESAR project conducted in cooperation with George Mason University (GMU), VA, USA.

The first paper is entitled *Web-Based Interfaces for Simulators of Coloured Petri Net Models* [54]. The paper has been published in *International Journal on Software Tools for Technology Transfer (STTT)*, Volume 3, Number 4, pages 405-416, Springer-Verlag, September 2001. The paper is contained in full in Chap. 6.

The second paper is entitled *Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic* [55]. The paper is joint work with Sajjad Haider from GMU and has been published in the proceedings

of *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 35-54, DAIMI PB–544, Department of Computer Science, University of Aarhus, 2001. The paper is contained in full in Chap. 7.

The third paper is entitled *Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets* [56]. The paper is joint work with Lee W. Wagenhals from GMU and has been published in *Formal Methods in Software Engineering and Defence Systems 2002*, *Conferences in Research and Practice in Information Technology*, Volume 12, pages 115-124, Australian Computer Society Inc., 2002. The paper is contained in full in Chap. 8.

Chapter 3 summarises the paper *Towards a Monitoring Framework for Discrete-Event System Simulations* [62]. The paper is joint work with Lisa Wells and will be published in the proceedings of *Workshop on Discrete Event Systems (WODES'02)*, IEEE, 2002. The paper is contained in full in Chap. 9.

Chapter 4 summarises the paper *Annotating Coloured Petri Nets* [61]. The paper is joint work with Lisa Wells and will be published in the proceedings of *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2002. The paper is contained in full in Chap. 10.

Chapters 2-4 are each divided into three sections. First an introduction and some background for the research is given. Next a summary of the main contributions presented in the paper(s) is given. Finally, related work is considered. Chapter 5 concludes on the work presented in this dissertation, and discusses ideas and directions for future work.

1.3.1 Reader's Guide

The reader is not required to have detailed knowledge of CP-nets and the corresponding tools. However, a basic knowledge of Petri nets will be assumed. Readers without such knowledge are advised to read [48] which gives a general introduction to CP-nets. The papers in Part II may be read in any order. However, the papers are organised in the order that appears to be most natural.

Readers with interest in web-based interfaces and simulators for influence nets may read only Chaps. 6-8, while readers with interest only in defence systems may read only Chap. 8. Readers with interest only in annotations and monitoring of CP-nets, may read only Chaps. 9-10.

Chapter 2

Web-Based Simulation of CPN Models

This chapter considers three papers documenting the part of the research in the CAESAR project that the author of this dissertation has participated in. The three papers are entitled *Web-Based Interfaces for Simulators of Coloured Petri Net Models* [54], *Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic* [55], and *Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets* [56]. Section 2.1 gives some background on the CAESAR project and contains a brief introduction to the results in the papers. Section 2.2 gives a summary of the main contributions of the three papers. Section 2.3 contains a discussion of related work.

2.1 Introduction and Background

When the work on this dissertation began in 1999, most uses of CP-nets in Design/CPN were conducted by people who are experienced with using CP-nets and the corresponding tools. Simulation results were most often interpreted directly from the markings displayed in the CPN tools. As a result, knowledge of CP-nets and CPN tools was often required to be able to use CP-nets.

In the CAESAR research project at George Mason University (GMU) in USA, CP-nets are used to simulate so-called *influence nets*[1] [90]. Influence nets are probabilistic state-equilibrium net models which are proposed to be used in the US military for modelling and evaluating military plans. Figure 2.1 contains an example of an influence net with six nodes modelling a military situation. Influence nets are well suited for modelling probabilistic behaviour. However, no time concept exists for influence nets, and it is not possible to execute influence nets. The consequence is that by using influence nets only, it is not possible to obtain a time ordered plan specifying when to conduct the various activities in the plan.

The researchers at GMU decided to define a translation from influence nets to CP-nets. The major advantage of translating influence nets to CP-nets, is

[1] In this overview paper we only use the term *influence nets* when referring to either influence nets or to timed influence nets with logics (TINLs). The reason is that in the overview paper the difference is not considered as important.

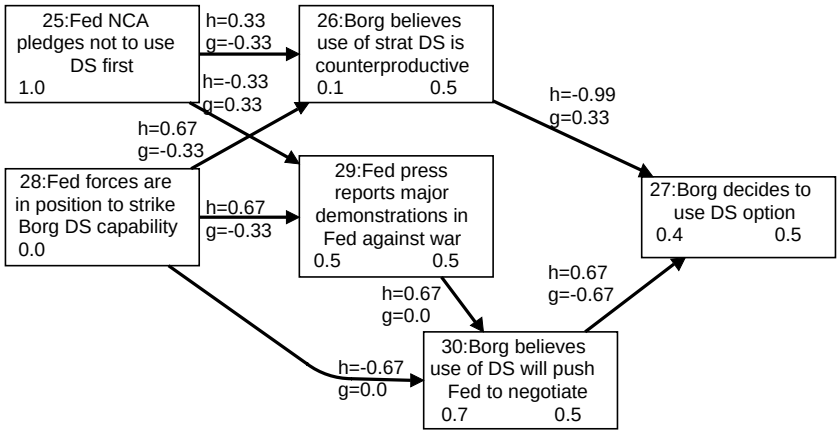


Figure 2.1: An influence net.

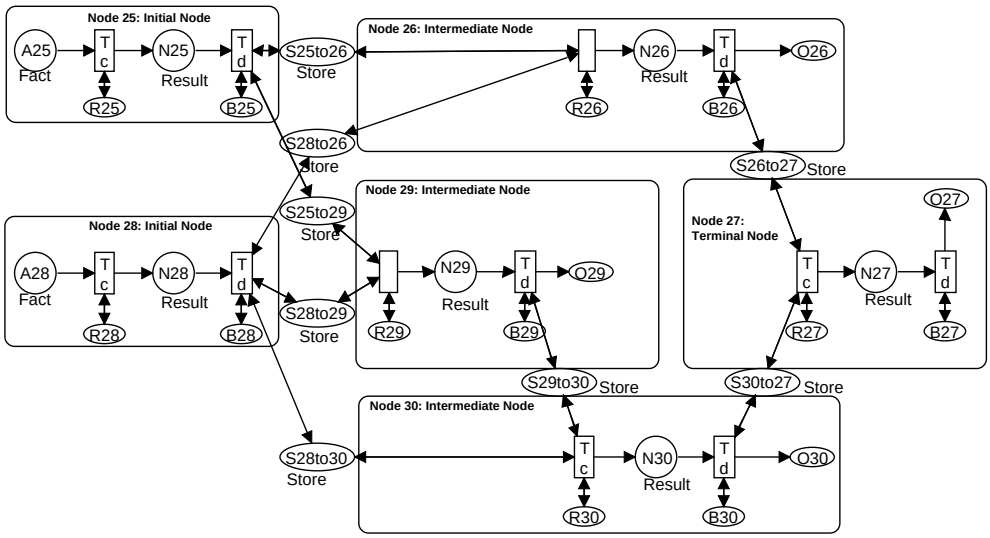


Figure 2.2: CP-net structure for the influence net in Fig. 2.1.

that the time concept for CP-nets can be used to add a time concept to influence nets. In addition, the translation makes it possible to simulate an influence net [105]. The researchers at GMU implemented a completely automatic translation from an influence net to a CP-net in Design/CPN [108]. The essential net structure of the automatically generated CP-net for the influence net in Fig. 2.1 is contained in Fig. 2.2. To apply the translation, a file specifying a concrete influence net is first exported from the tool used to create the influence net. Next, an algorithm within Design/CPN uses the file to automatically generate places, transitions, arcs, and declarations to create a full CP-net. As seen from Figs. 2.1 and 2.2, the CP-net has a net structure which is very similar to the net structure of the influence net. The similar net structure makes it easy to relate the two nets to each other, if necessary.

The turn-around time for applying the translation method is relatively long. Even though the process is entirely automatic, it takes minutes to translate an

influence net to a CP-net and afterwards to generate the corresponding simulator code. Efforts was made by researchers at GMU to *optimise the translation algorithm*, but for non-trivial influence nets, most time was spent internally in DESIGN/CPN for generating the simulator code, and therefore they had limited success with the optimisation.

The people going to apply the translation and afterwards simulate the CP-net are military analysts without knowledge of CP-nets. It would therefore be useful for these people, if it is possible to create a *domain-specific graphical user interface* (GUI) which uses concepts from their application domain instead of CPN terminology. In other words, the GUI of the CPN tool should be hidden behind a customised GUI which is created to fit the given application domain.

2.2 Main Contributions

The research presented in the three papers [54, 55, 56] discussed in this chapter is divided in two main areas. First we describe, a method which has been developed to create domain-specific GUIs for CPN simulators. A general description of how to apply the method in a CPN tool can be found in [54], and practical use of the method in the CAESAR project is presented in [56]. Secondly, we focus on the automatic method for translating influence nets to CP-nets. The translation has been modified to speed up the overall turn-around time of the process. This has been documented in [55, 56]. In the following we will discuss these two main areas separately.

2.2.1 Domain-Specific GUIs for CPN Simulators

As part of the CAESAR project, the author of this dissertation participated in designing and implementing a domain-specific GUI for the CPN simulator of influence nets. Instead of using the GUI of the CPN tool, the interaction with the CPN simulator is done via a GUI which uses concepts well-known for the people with knowledge of the influence net, and thereby the GUI completely hides the CPN-specific concepts. The technology chosen for creating the GUI and for communicating with the CPN simulator is based on the web technology available in the year 1999 when this work was conducted. Ordinary *HTML forms* [102] are used for specifying input parameters to the CPN simulator, while so-called *common gateway interface (CGI) scripts* [35] are used to invoke the CPN simulator. A CGI script is essentially a program that can be executed from a web page, and then dynamically produces a new web page.

In the CAESAR project, a complete web site has been developed, intended to support simulation of influence nets for operational planning in the US military. The web site is highly dynamical due to the fact that most of the web site is generated automatically before or while simulating influence nets. As part of this generation, simulation results are stored as HTML documents with graphics, text, and raw data. These simulation results can then later be accessed from the web site and used for further analysis, e.g. by generating comparison profiles for comparing simulation results from different simulations of an influence net. Figure 2.3 depicts an example of an automatically generated web

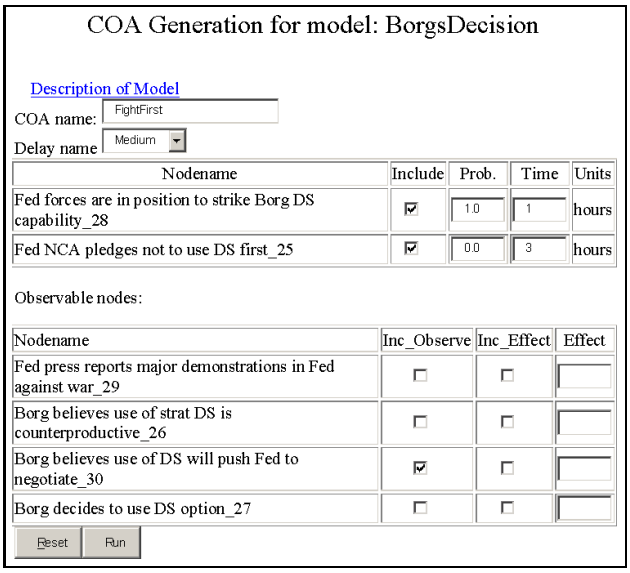


Figure 2.3: Web page to specify input to a CPN simulator for influence nets.

page containing an HTML form used to input a number of parameters for the CPN simulator of influence nets. Notice that the text in the HTML form is taken from the influence net in Fig. 2.1. When the HTML form is submitted the parameters are transferred to the web server and given as parameters to the CGI script. The CGI script initialises the CPN simulator with the parameters and runs a simulation. During the simulation, data is collected and is used to generate simulation results. These simulation results will be both displayed in the web browser and saved in files for later use. Figure 2.4 shows an example of some of the graphical output which can be automatically generated as simulation results. The graphs shows the changes in the probabilities as a function of time for three influence net nodes. This graph gives a very compact overview of the changes in probabilities. Several other similar graphs are generated to give more detailed information of the probabilities of the individual nodes.

The web environment has been evaluated in the war game *Naval War College Global 2001* by US military people from the Office of Naval Research and by researchers from GMU [106, 107]. The purpose of the war game was to gain insight into the potential of different applications supporting course of action (COA) development and evaluation, and to provide insight into how the developed concepts can be incorporated into real-world operational environments. As part of the war game, models of complete battle plans were built and evaluated using the web environment. The war game showed that the web environment is indeed useful in practice by making it possible for people without knowledge of CP-nets to translate influence nets to CP-nets, perform simulations of CP-nets, and to interpret simulation results via the domain-specific GUI.

From the experiences with creating the GUI for the CAESAR project, a general method for creating domain-specific GUIs for CPN simulators using HTML forms and CGI scripts has been defined and documented in [54]. We

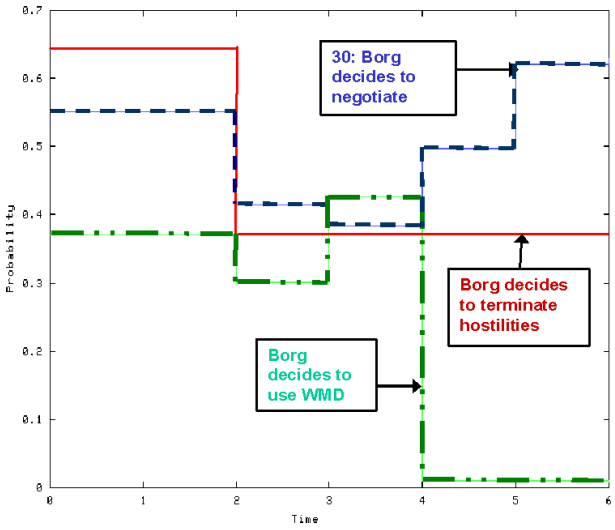


Figure 2.4: Graph generated as part of the simulation output.

discuss this general method in more details in the following.

There are a few essential requirements for a CPN tool to be controlled via HTML forms and CGI scripts. First of all, the CPN tool should be able to start a simulation of a specific CPN model without requiring a user to interact with the tool-specific GUI. For example, it should not be necessary to use dialog boxes in the CPN tool to start a simulation – instead it should be possible to automatise such actions in, e.g. a script. Therefore, the CPN tool should allow a user to create scripts for e.g. retrieving input parameters, saving files, and printing to standard output. The reason for these requirements is that because a CGI script is invoked via an HTML form, the CGI script must be able to control everything related to reading input from the HTML form, setting the initial state of the CPN simulator, starting the simulation, and producing simulation results to be included in the resulting HTML document which will be displayed in the web browser.

In Design/CPN it is indeed possible to customise control of the tool. A so-called *batch script* [57] can be used to specify a sequence of commands that must be executed when the CGI script is invoked. The programming language SML is used to specify batch scripts. Due to the fact that SML is used both as inscription language in Design/CPN and to implement the CPN simulator, it is possible to access all the functionality of the CPN simulator from a batch script. The modeller who creates the CPN model is also expected to have a certain knowledge of SML, which should make it rather easy for the modeller to create a batch script based on examples of other batch scripts.

Another issue which makes CGI scripts easy to produce in Design/CPN is that it is possible to isolate the CPN simulator and batch script from the GUI of Design/CPN. This means that it is easy to create a CGI script as a stand-alone program containing only the batch script and the CPN simulator for the specific CP-net.

HTML forms and CGI scripts are only suitable for non-interactive simulations due to the fact that all parameters are given before starting a simulation. In other words, using HTML forms and CGI scripts, it is not possible to interact with the CPN simulator during a simulation. However, it is our experience that often all parameters are available before the simulation is started, and therefore the approach presented here is sufficient.

2.2.2 CPN Simulators for Influence Nets

In this section we consider the CP-nets for simulating influence nets. As mentioned above, the translation developed by researchers at GMU creates a CP-net that is specific for a given influence net. Thus any time a new influence net has to be investigated, a completely new CP-net must be generated and the CPN simulator for this specific CP-net must be generated. We denote this CP-net by the *fixed CP-net* to reflect the fact that the CP-net is fixed to a specific influence net and cannot be used for simulating other influence nets[2]. In the application envisioned in the CAESAR project, it is likely that influence nets changed frequently. This requires time to conduct the translation and generation of the CPN simulator. Users of the web environment must therefore first instruct the server to generate and compile the fixed CP-net before being able to simulate it.

In [55] we have developed an alternative approach. Instead of generating a new CP-net for each influence net, we have developed a *generic CP-net*[3]. Thus, while a fixed CP-net has net structure, inscriptions, and colours fixed to the specific influence net, the generic CP-net has only the initial marking depending on the specific influence net. In other words, everything in the CP-net is constant for all influence nets – except the values of tokens. This means that the generic CP-net can be fixed to a specific influence net by setting an initial marking to reflects the structure of the specific influence net. Using a batch script as discussed in the previous section, a set of parameters determining the initial marking are specified automatically from the same influence net file as used for generating a fixed CP-net.

Let us consider how the generic CP-net has been created. It is largely inspired by the automatic translation for fixed CP-nets. The translation to fixed CP-nets maps each node in an influence net to small CPN-subnets, as indicated in Fig.2.2. All these subnets are then connected via places constituting the interface between the subnets. These interface places represent the arcs connecting nodes in the influence net. So the complete fixed CP-net consists essentially of several structurally equal subnets and a number of connecting places. The generic CP-net has been created by folding all these subnets to a few general subnets. To know which subnet (i.e. node in the influence net) a token belongs to, the colour sets have been extended with a node id to specify which node in the influence net the token corresponds to. Instead of modelling the arcs in the influence net with interface places in the CP-net, the arcs connecting the influence net nodes are modelled using tokens specifying the source and

[2] In [55] the term *unfolded CP-net* is used instead of the term *fixed CP-net*.
[3] In [55] the term *folded CP-net* is used instead of the term *generic CP-net*.

destination of the nodes in the influence net. In this way it is possible to model any influence net by setting the initial marking of the generic CP-net.

For a given influence net, the fixed and generic CP-nets are expected to give the same simulation results. This has been validated using state space analysis for samples of influence nets. For each sample of influence nets, the corresponding fixed CP-net was generated and the initial marking of the generic CP-net was set to reflect the influence net. Then the state spaces were generated for each of the CP-nets and compared using various state space analysis techniques.

In [56] we have considered the performance of the obtained influence net simulator using the generic CP-net compared to using the fixed CP-net. In the generic CP-net, the number of tokens on each place increases linearly with the number of nodes in the influence net. As a consequence, the enabling computation of the Design/CPN simulator is slowed down for larger influence nets, and can therefore potentially result in very slow simulations (minutes or even hours). Due to the fact that the generic CP-net was developed mainly to improve the effectiveness of the overall method, it is not acceptable that the simulation time for the generic CP-net increases so much that the total time is larger than for the fixed CP-net. The solution for reducing the simulation time was to use the new simulator of Design/CPN which has been optimised for CP-nets with many tokens [77]. Using the new simulator, the total time used for applying the generic CP-net is in most cases reduced to a few seconds which is fully acceptable for web-based simulations.

2.3 Related Work

Web-based simulation is getting more and more attention during these years. Almost every computer is connected to the Internet. This makes it easy for many people to access programs on web servers. Therefore, it is obvious to investigate the new potential advantages from conducting web-based simulations. The rapid advances in web technology has the consequence that the web technology of today will be considered to be old-fashioned in a year from now. Therefore, the current research on web-based simulations may be considered as pioneering in the field.

In the following, we will consider work related to the main topics discussed in Sect. 2.2. Translations from other languages to CP-nets will be discussed. A few simulation tools where the GUI can be customised, and other ways of creating domain-specific GUIs, will also be discussed.

Translating other Languages to CP-nets

The fact that CP-nets can be executed and have a formally defined semantics is one of the reasons why CP-nets are very popular. Languages without a formal semantics and without the ability to be executed, but where these properties are needed, are sometimes translated to CP-nets. In the CAESAR project presented above, influence nets are created in an influence net tool. Afterwards, the influence net is automatically translated to CP-nets where time is added,

and thereby CP-nets defines the formal semantics of influence nets. Finally, the influence net can be executed.

A similar idea has been applied in a project from 1991 described in [78]. A structured analysis and design technique (SADT) was used to create CPN models of a nuclear waste programme. A tool supporting the SADT technique was used to create workflow descriptions of the activities to be performed in the waste management programme as a set of so-called IDEF0 diagrams. The SADT model was then translated into a CP-net. In contrast to the CAESAR project, the translation was only partially automatic meaning that inscriptions had to be added manually by CPN modellers. The SADT tool was operated by people with knowledge of SADT – but without knowledge of CP-nets. The manual part of the translation from SADT to CP-nets and the simulations was conducted by CPN experts.

The SADT project had a similar need for presenting simulation results to the SADT experts using concepts from their specific domain. For this purpose ad-hoc diagrams were created containing no CPN concepts but using solely concepts from the SADT environment.

The CP-nets created from SADT diagrams has net structure and inscriptions which are fixed to the concrete diagram. This is similar to the original translation method from influence nets to fixed CP-nets. As mentioned, the translation from SADT diagrams to CP-nets includes manual work by CPN experts where inscriptions are added and unspecified behaviour is resolved. By creating a generic SADT diagram simulator and a domain-specific GUI for this simulator, it would maybe be possible to supply the manually added information using domain-specific concepts. Likewise the simulation and post-processing of simulation results could be generated automatically. The consequence is that the CPN experts are not needed in the process. Based on the SADT project described above and an earlier similar project [84], a work flow tool [72] was later developed making the translation completely automatic for a restricted set of SADT models.

ExSpect [1, 25, 26] is a workflow modelling and simulation tool supporting a variant of CP-nets. Facilities have been created in ExSpect to translate from workflow modelling tools like, e.g. PROTOS [4], to ExSpect models. In the process of creating an ExSpect model from a PROTOS model, the translation uses predefined building blocks from a workflow library of ExSpect building blocks.

The unified modelling language UML [29] is a standardised graphical modelling language which is widely used for object-oriented modelling. Researchers at George Mason University, USA, have defined a translation from so-called UML activity diagrams to CP-nets to obtain an executable UML specification [104].

SDL [11] is another visual modelling language which is widely used in the tele communication area. Research on a fully automatic translation from SDL diagrams to variants of Petri nets has been conducted in [41].

Domain-Specific GUIs for CPN Simulators

The fact that the GUI of the DESIGN/CPN tool uses CPN-specific concepts has been challenged in several projects. Several efforts have been made to interact with a simulation and to present results using domain-specific concepts.

In 1995 a project with participants from both industry and universities developed a CP-net of an alarm system [86]. During the project a graphical animation library called MIMIC was developed for DESIGN/CPN [87]. During simulations, the MIMIC library animates the simulation within DESIGN/CPN by showing, moving, and hiding graphical objects in a drawing of the building containing the alarm system. The user can inspect the state of the simulator via graphical objects instead of investigating tokens in the CP-net. By clicking on the icons, the user can provide input to the CPN simulator. The project illustrated that the MIMIC library made it possible for people without knowledge of CP-nets to interact with a DESIGN/CPN simulator of CP-nets using domain-specific graphics.

The need for domain-specific presentation and interaction with CPN simulators was also illustrated in a project at Nokia research centre in Finland in 2001 [66, 67]. The goal of the project was to model feature interactions in mobile phones using CP-nets to test feature interactions. Interaction with the CP-net was done via a picture of a mobile phone. The picture was instrumented with MIMIC code such that interaction could be conducted by clicking and looking at the picture of the mobile phone. The consequence was that people without knowledge of CP-nets could experiment with different features to test if the phone behave as expected.

Message sequence charts (MSCs) [40] is a telecommunication standard for specifying and visualising communication patterns in protocols. During a telecommunication project [9], a MSC library for DESIGN/CPN was developed for automatic generation of MSCs. Several projects [16, 66] have proved that such automatic generation of MSCs to visualise communication patterns during a single simulation of a CP-net is very useful. The MSCs are useful for discussing the behaviour of different scenarios without having to show the details of the CP-net. In the project discussed in [66] it was illustrated that it is useful to generate MSCs with different levels of abstraction, i.e. some MSCs give a very detailed description while others describe the communication patterns at a more abstract level.

The workflow tool EXSPECT has recently isolated the core parts of the tool engine in a so-called *component*. A component is a unit with contractionally specified interfaces and explicit context dependencies, and it is subject to composition by third parties [96]. Essentially, the EXSPECT component makes it easy to develop applications with EXSPECT inside to provide the workflow engine. Technically, the application programming interface (API) of the EXSPECT component provides a number of methods which can be accessed by the application to control the component. Related work for DESIGN/CPN has been done in a Master's thesis (co-supervised by the author of this dissertation) [81]. A simple simulation control protocol was implemented on top of the communication protocol TCP/IP [22]. Based on the control protocol, a JAVA APPLET [42]

was used to visualise and interactively control a Design/CPN simulation of the well-known dining philosophers example. This work was the basis for the development of the Comms/CPN library for Design/CPN [30].

It is not only simulation tools that can take advantage of domain-specific GUIs. For analysis tools used for e.g. state space analysis, much effort is put into automatising the use of the methods and to hide as many irrelevant technical details as possible. The purpose is to make it possible for people to use these analysis methods in practice even though they do not have detailed knowledge of the mathematical background of the analysis methods and algorithms. As an example, the Design/CPN State Space Tool [48, 18] is able to generate a report containing several model-independent analysis results. No knowledge of state space analysis is necessary to generate this report which relies on relatively complex mathematically defined analysis methods. The contents of the report can also be understood with only limited knowledge of the analysis concepts. In [45, 65, 64] permutation symmetries for CP-nets are exploited to generate reduced state spaces. The user is required to provide the permutation symmetries. Afterwards, a consistency check must be performed to check that the permutation symmetry is actually consistent for the given CP-net. As part of the project [64], a semi-automatic check for consistency was developed in Design/CPN to make the method more accessible to people without detailed knowledge of the analysis method.

In [112] work very similar to the work presented in Sect. 2.2 and in [56] is presented. A tool, called COAST, based on CP-nets and Design/CPN supporting operations planning in the Australian Defence Force has been developed. The core part of the COAST tool consists of a CP-net which models the execution of tasks. During planning, the CP-net is instantiated with concrete tasks for execution and analysis in the same way as our generic influence net CPN simulator from the CAESAR project. The COAST tool does not support web-based simulations. The tool is implemented as a client-server architecture where the client consists of a domain-specific GUI implemented in Java [42], while the server uses the Design/CPN State Space Tool to analyse the given model. The GUI supports creation of models and display of analysis results. This means that the user does not need to switch applications when modelling and analysing a model, like it is necessary in the CAESAR tool.

Other variants of Petri nets have also identified some of the advantages of web-based interfaces to Petri net tools. In [39, 7] a tool named WebSPN supporting stochastic Petri Nets is described. The GUI of the tool is implemented as a Java Applet. Java Applets can be included in a web page, and started when the web page is accessed. The GUI presented in the paper is Petri net specific, and it is not discussed whether domain-specific GUIs are supported.

In [8], web-based simulations of models based on so-called fuzzy sets [111] is described. They stress the fact that end-users can take advantage of interacting directly with the model using a domain-specific web-based GUI to obtain better validation of a model. The alternative is that the end-users have to communicate with model-experts by explaining their needs for results to modellers which can then conduct simulations, and finally explain the simulation results

to the end-users using domain-specific terms. The application is implemented as a Java Applet with both the GUI and the simulation program contained in the Java Applet. This is in contrast to the CAESAR simulation environment where the simulator is running on the web server, and only the GUI is downloaded to the client.

A general discussion and overview of different web technologies supporting web-based simulations is contained in [75]. Technologies in the spectrum from download-and-run applications to very thin clients are briefly discussed. Concrete examples of technologies are Java Applets, Java Servlets, Java Beans, and CORBA. The simulation tool discussed in [75] is called JSIM and it supports creation of general simulation packages not necessarily based on Petri nets. The general idea is that a simulation model can be turned into a simulation component, and several simulation components can be combined to a large simulation model which can both be simulated from web-applications and integrated into other applications. However, it seems like the tool lacks a formal foundation, and therefore do not support formal analysis.

Chapter 3

Monitoring Framework

This chapter treats the paper *Towards a Monitoring Framework for Discrete-Event System Simulations* [62] which is written in cooperation with Lisa Wells, University of Aarhus. Section 3.1 gives a brief introduction to the results in the paper. Section 3.2 gives a summary of the main contributions in the paper. Section 3.3 contains a discussion of related work.

3.1 Introduction and Background

Simulation of discrete-event systems may be conducted to obtain knowledge about the behaviour of the modelled system. During simulations, it may therefore be necessary to examine the states and events of the system and then periodically extract or collect data from the states and events. Collecting data may be conducted by augmenting the model itself with ad-hoc code for collecting data or by using integrated facilities in the simulation tool. In other words, either the model itself or the simulation tool can be augmented with data collection facilities.

Integrated tool-support for collecting numeric data is present in the Design/CPN tool, and is called the Design/CPN Performance Tool [58]. The Design/CPN Performance Tool was designed and implemented by the authors of the paper as part of their Master's thesis [59], and the work presented here is a continuation of this work. The Design/CPN Performance Tool makes it possible for a user to automatically extract data from complex token values and binding elements in a CP-net during simulations. By accessing a binding element it is possible to extract the values bound to variables when the corresponding transition occurs. Furthermore, data can be collected from places in the entire CP-net which means that it is possible to access the marking of the entire CP-net when defining how to collect data. To define what data should be collected during a simulation, the modeller must define *when* and *how* data should be collected. In the Design/CPN Performance Tool this is done by specifying two functions separately from the CP-net. A predicate function named `check` is invoked after each step in a simulation. If it evaluates to true, then a function named `observe` is invoked to extract or calculate the actual value to be collected. The tool uses these functions to collect data, and it

automatically saves the collected data in files or uses it for computing statistics of the observed numbers.

When designing the Design/CPN Performance Tool, the intention was to create a tool supporting *simulation based performance analysis* [60]. During experiments, it became clear that the collected data can be used for several other purposes than performance analysis. Examples are visualisation of the behaviour using e.g. message sequence charts [71], transmission to external processes using TCP/IP communication [81, 30], and for stopping the simulation when a certain state is reached.

In addition to using collected data for several purposes, we realised that the tool can also be used in the process of post-processing the collected data. In the three examples mentioned above, the tool can be manually customised by the modeller to invoke the libraries used for visualisation, communication, and simulation control after having collected each piece of data. These additional possible uses are due to a general design of the Design/CPN Performance Tool and the possibility for the user to customise the facilities. These findings have served as a basis for generalising the Design/CPN Performance Tool to a so-called monitoring framework for discrete-event systems.

3.2 Main Contributions

Based on our experiences with implementing and using Design/CPN, we have observed that the design and implementation of efficient 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. This means that when a new way is needed for processing information from a simulation, then a new mechanism and a new way to define how to extract the information is invented and implemented in the tool. Some of these ad-hoc methods are directly reflected in the models, e.g. via added events that are used solely to extract information. Adding such events may introduce errors into the models. From our point of view, there should be a clear distinction between modelling the behaviour of a system and monitoring the behaviour of the model.

Even though the information extracted during a simulation may be used for various purposes, the way the information is extracted can be unified. This means that it is possible to create a common and general mechanism for defining how to extract information from a model, and this is the main objective of the research presented in the paper.

In the paper, we use the concept *monitor* to refer to a 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. The concept, *monitoring* is defined in the paper as any activity related to observing, inspecting, controlling, or modifying a simulation of a model. For example, a monitor in a model of a communication protocol could inspect the events during a simulation of the model and update a message sequence chart each time an event appears that corresponds to the transmission of a message. Furthermore, we make a distinction between *modifying monitors* which can influence the state

of a model during a simulation and *inspecting monitors* which are unable to alter the state of a simulation model.

The goal of the research presented in the paper was to develop a 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 have developed a general and flexible framework that can be used for defining many different types of monitors. It is our experience with practical use of ad-hoc monitoring techniques in Design/CPN and our experience with the Design/CPN Performance Tool that is the motivation for developing the monitoring framework. The monitoring framework has evolved as a generalisation of the data collection facilities in Design/CPN Performance Tool to support other activities than collecting data for performance analysis. The goal was also to extend the scope of the framework from CP-nets to discrete-event system models in general.

The main observation leading to the definition of the monitoring framework is that in the Design/CPN Performance Tool the user can define *when* and *how* to extract data, and the tool then decides *what* to do with the data. In other words, the user cannot define what to do with the data. By allowing the user to define what to do with the data after having collected it, it becomes possible to use the tool for several purposes not related to data collection. The *what* is specified by allowing the user to create a function named `act`, analogously to the functions `check` and `observe` specifying when and how to monitor. Theerefore, the most essential parts of a monitor are these three functions:

i. *Check:* When or how often should the monitor be invoked to make an observation.

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

iii. *Act:* Based on the observation, what action should be made.

Often some of these three functions are independent of the model. Therefore, it is possible to predefine some of these functions to support a certain use. For example, a *data collection monitor* may be predefined by creating an `act` function which takes care of saving the observation from the `observe` function in a file or using it for updating statistics. The user then only needs to manually define the `check` and `observe` functions which are the only functions depending on the model. Likewise, a *simulation stop monitor* may be predefined by creating an `act` function which sets a stop flag in the simulator causing the simulation to be stopped. The user then only needs to manually define the `check` function, while leaving the `observe` empty due to the fact that no value should be observed to stop the simulation; it is sufficient that the `check` function specifies when to cause the `act` function to be invoked.

Being able to predefine some parts of a monitor as described above, makes it possible to support a design requirement saying that often used monitoring must be easy to do, while rarely used and complex model dependent monitoring should be possible. Therefore it is possible to create a tool-box of monitors which can easily be used by a user with minimal effort. However, if the modeller

has a very special kind of monitoring need, more work may have to be done manually than if the kind of monitoring is often used and independent of the given model.

It is our experience that the monitoring framework makes a clear separation of monitoring facilities from other facilities of a discrete-event system simulation tool. The framework is designed such that it can be used to create a monitoring component to be plugged into a simulation tool and thereby extend the given tool with extra monitoring facilities.

3.3 Related Work

Monitoring facilities are available in several different tools. However, monitoring is rarely identified explicitly as a general methodology but is rather implicitly contained in specific facilities like, e.g. performance analysis facilities. In the following, we will discuss and relate some of these tools to the monitoring framework discussed above.

Monitoring in Petri Net Tools

For Petri net tools, monitoring is most often based on simulations. Some tools only contain fixed built-in facilities for monitoring, while others allow some kind of customisation of the facilities.

Simulation based performance analysis of Petri nets is one field where monitoring is particularly widespread. The purpose of simulation based performance analysis is to simulate a model to reveal information about the performance of the system in question. In practice, it is conducted by monitoring events and states during simulations.

The tool UltraSAN [97, 98] is based on Stochastic Activity Nets (SANs) [73, 79, 91]. SANs are basically low-level Petri nets with stochastic time and a few other extensions which are not important for this discussion. UltraSAN supports user-defined performance measures meaning that the user can specify performance measures. This is done by specifying a so-called *reward function*. A reward function is very similar to a data collection monitor in our monitoring framework. A reward function is defined by two components: a *rate* reward function and an *impulse* reward function. The rate reward function is used to extract data from markings of places, while the impulse reward function is used to extract data when activities (transitions) complete.

Rate reward functions are defined by means of a check function and an observation function like for a monitor. These rate reward functions can refer to the markings of different places. SANs are low-level nets, so only the sizes of the markings are returned by the marking functions. This is different from what is possible when considering CP-nets. In CP-nets, tokens are coloured, i.e. it is possible to distinguish between tokens with different colours in one place.

Impulse reward functions are associated with transitions. When defining an impulse reward function it is possible to specify that for a specific transition occurrence, a constant must be added to the reward function. This means that

it is not possible to compute a value based on e.g. markings, when a transition occurs and then return the value, as proposed in the monitoring framework.

In UltraSAN, the data collected by the reward functions are used internally, e.g. to maintain statistics and to generate graphs. When considering the monitoring framework, this means that the act function is fixed to a specific use. Therefore, the user is not able to affect how the data should be used, except from choosing among the predefined alternatives.

The tool GreatSPN [14, 33] supports generalised stochastic Petri nets (GSPNs) [2]. In short, GSPNs differ from CP-nets in that they have uncoloured tokens and three different types of transitions (deterministic, exponential and immediate, which refer to the distribution of firing delay). GreatSPN is a prevalent tool that is widely used for creating and conducting performance analysis of GSPNs. The analysis may be conducted using both simulation and analytic analysis. Like for UltraSAN it is possible to create user-defined performance measures in GreatSPN. It is also possible to monitor several places in the model using a single performance measure. However, there is not an explicit separation between specifying when and how to monitor as proposed in our monitoring framework.

The CPN/DESIR [85] is a tool for debugging and simulating CP-nets. With this tool, which is integrated in the CPN-AMI tool [70], the user can associate a script with a transition. When a transition is fired, the associated script is interpreted. A script is a sequence of elementary instructions. Among others, the instructions can be used to do data extraction by printing or saving e.g. the marking of a place. The check for when to execute an action is specified by associating the script to certain transitions. The contents of the script defines what to monitor and what to do with the monitored data. This is very similar to how code-segments are used in Design/CPN.

The Design/CPN State Space Tool [47] supports state space analysis of CP-nets. The analysis is conducted by investigating the state space for the given CP-net. This investigation can be considered as monitoring the state space. In the Design/CPN State Space Tool so-called *queries* are used to analyse state spaces. A query is a function which can be mapped on a state space for extracting certain information from both markings and binding elements. Essentially, a query is specified by means of three functions. A *predicate* function specifies which states or binding elements should be investigated, an *evaluation* function specifies how to extract information from the given state or binding element, and finally, a *combination* function specifies what to do with the extracted information. The Design/CPN State Space Tool allows user-defined queries for model-dependent analysis and standard queries for model-independent analysis. The standard queries are often-used queries that can be applied to any model without or with limited customisation. These concepts are very similar to the concepts of the monitoring framework. The three functions used to define a query are similar to, and have essentially the same domain and range as the corresponding functions for monitors. Likewise, monitors and queries can both be user-defined or predefined.

Other Tools with Monitoring Facilities

The widget observation, simulation, and inspection tool (WOSIT) [110, 13] by MITRE [76] is a software instrumentation tool for X-Windows applications. The purpose of WOSIT is to support embedded training systems for end users by monitoring and interacting with the GUI of any X-Windows applications. In general, when a software application is instrumented, internal states and processes of that application become accessible from the WOSIT tool. WOSIT supports instrumentation of the GUI of an X-Windows application in a manner that requires no modifications to the source code of the instrumented application. WOSIT acts as an intermediary between the instrumented application and a client application. It enables the client application to observe the user's manipulations of the GUI of the target application (e.g. button presses, changes to text fields), and to inspect states of the GUI (e.g. the contents of text fields and the state of radio buttons). In addition, WOSIT enables the client to initiate certain actions on the GUI of the target application. This is very similar to how monitors in our monitoring framework interacts with a model and a simulator. To use a monitor, no modifications need to be made to the model itself just like no modifications should be made to an application to be monitored by WOSIT.

Another tool which monitors and interacts with an application is the Microsoft Office Assistant known from e.g. Microsoft Word [74]. This program monitors GUI interactions performed by the user, and proposes help if it seems like the user needs help. Microsoft also provide other types of monitoring programs. For example, accessibility of programs for people with disabilities is improved by a screen-reader for blind people. This program can access the text contained in an application and interpret it to voice.

Frameworks and Design Patterns

One of the purposes of creating the monitoring framework was to provide a uniform design for implementing future monitors of various kinds. This was motivated by the fact that our experience was that data collection monitors in Design/CPN Performance Tool are sufficiently general to be used for other purposes and for other kinds of discrete event systems than CP-nets. Thus, we wanted to reuse the design for many purposes in the specific context of discrete event systems. In fact, a *framework* is characterised in [31] as a set of cooperating classes that make up a reusable design for a specific class of software, and it captures the design decisions that are common to its application domain. This fits very well with our monitoring framework.

If we turn to reusing a design in different contexts, so-called *design patterns* are often used [31]. A design pattern considers a problem which occurs over and over again in an environment, and then describes the core of a solution to that problem in a way that can be used in any context. Thus, a design pattern gives a general description of communicating objects and classes that have been customised to solve a general problem in a particular context. In contrast to a concrete framework, a concrete design pattern can be applied in

different application domains.

Actually, the monitoring framework is intended to be realised using a design pattern, called *observer* (p. 293 in [31]). The intent of the observer design pattern is to define a one-to-many dependency between objects so that when one object (called *subject*) changes state, all its dependents (called *observers*) are notified and updated automatically. If we consider the state of the simulator to be a subject, then the individual monitors will be the observers. When the state of the simulator changes, only the monitors interested in that particular change should be notified.

Research on design patterns for Petri nets is contained in [34, 80] with focus on collecting and distributing Petri net modelling knowledge in the form of building blocks and pattern descriptions, in order to make Petri nets more popular for the use in industrial sized applications. The authors of the papers [80, 34] believe that the increasing use of patterns in ordinary software engineering domain may be transferred to Petri net modelling and thereby improve the use of Petri nets as part of software engineering. Design patterns for workflows have been investigated in [101, 100]. We believe that design patterns are very useful to make it possible for people with limited experiences to create models. However, if a model is going to be analysed using state space analysis, precautions should be taken for creating huge models by composing a set of building blocks. If not, then a state explosion is very likely to appear.

Chapter 4

Annotating Coloured Petri Nets

This chapter treats the paper *Annotating Coloured Petri Nets* [61]. Section 4.1 gives a brief introduction to the results in the paper. Section 4.2 gives a summary of the main contributions of the paper. Section 4.3 contains a discussion of related work.

4.1 Introduction and Background

Coloured Petri nets (CP-nets) can be used for fundamentally different purposes like functional analysis, performance analysis, and visualisation. To be able to use the corresponding tool extensions and libraries, practical use has shown that it is sometimes necessary to include extra auxiliary information in the CP-net. An example of such auxiliary information is packet delay 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 use of the same basic CP-net.

The resource allocation system CP-net in Jensen's volumes on CP-nets [44, 45] perfectly illustrates the need for several slightly different versions of a CP-net. Two of these versions are contained in Fig. 4.1. Figure 4.1(a) contains a basic CP-net which is suitable for state space analysis due to the finite state space. Figure 4.1(b) contains the basic CP-net extended with process counters for the `p` and `q` processes. The extended CP-net has been obtained by augmenting the process colour set `U` in the basic CP-net with an integer counter, and by modifying the arc expressions and initial markings to reflect the slightly modified or extended behaviour. Here it is important to notice that the addition of the process counter to the basic CP-net has not restricted the behaviour in the sense that if a transition is enabled in a reachable marking in the basic CP-net, then the same marking will be enabled in the extended CP-net – independently of the value bound to the cycle counter. In other words, every reachable marking in the extended version will be identical to a marking in the basic version when the cycle counters are removed. In fact, this has been proved in [45].

The fact that a CP-net – like the basic resource allocation CP-net presented

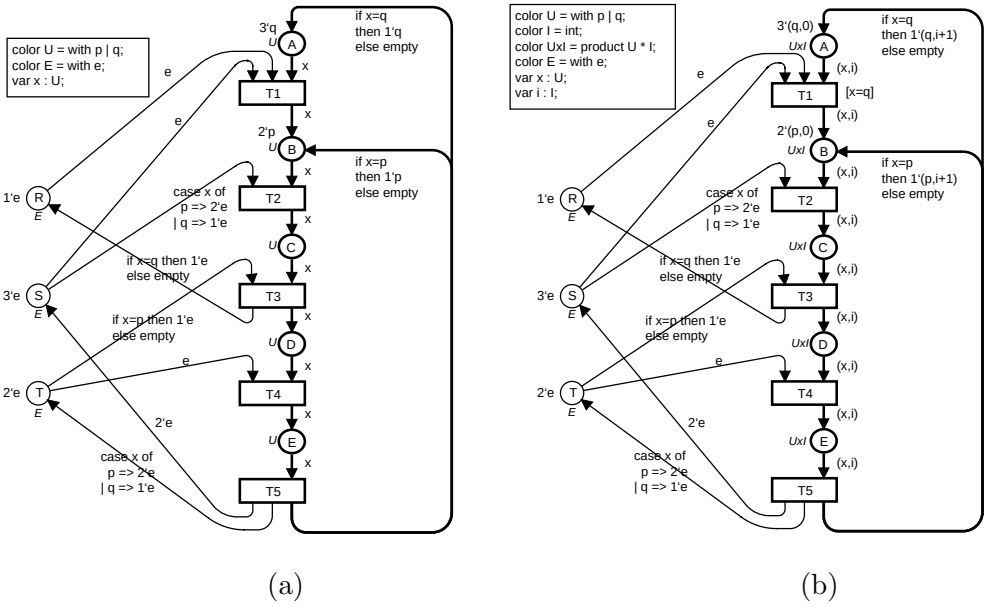


Figure 4.1: Different CP-nets modelling a resource allocation system. (a) The basic CP-net. (b) The basic CP-net extended with a process counter.

above – can be used for different purposes means that it is desirable if the tool extensions and libraries can be used without modifying or extending the CP-net itself. For example, it should be possible to use a CP-net for e.g. performance analysis, without having to add extra places, transition, and extend colour sets with the only purpose of supporting data collection. In particular, when considering industrial sized CP-nets it is very useful to be able to use a single CP-net for several purposes without having to maintain different copies of the same basic CP-net. From our point of view, it is best if the auxiliary information is not integrated into colour sets and arc inscriptions of a CP-net, but is kept separately, so that it is easy to disable this information if the CP-net is to be used for something else.

In Chap. 3 a monitoring framework was presented. It defines a consistent way to separate the code used for monitoring from a CP-net. Instead of integrating the monitoring facilities into the CP-net, the monitoring code is defined separately from the CP-net and is then able to refer to the elements of the CP-net. By separating the monitoring code from the CP-net it is often not necessary to add extra places, transitions, and net structure to a CP-net for the specific monitoring purpose. However, even though the monitoring framework is used, it is still often necessary to modify colour sets and arc-inscriptions in a CP-net to be able to use a CP-net for different kinds of analysis. This issue has been addressed in the paper discussed in this chapter.

4.2 Main Contributions

The paper proposes that auxiliary information is not integrated into colour sets and arc inscriptions of a CP-net, but is kept separately. The separation makes it easy to disable this auxiliary information if a CP-net is to be used for another purpose. A method is proposed which makes it possible to augment tokens with auxiliary information, called *annotations*, 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 – like the process cycle counter in the resource allocation system discussed in the previous section.

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. Annotations are defined in a manner similar to how timed CP-nets are defined in the sense that some tokens carry time stamps while others do not. 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*. An annotated version of the basic CP-net in Fig. 4.1(a) is contained in Fig. 4.2(a) with an annotation layer modelling the cycle counter. The annotations in the annotation layer are black, while the CP-net itself is grey. Notice that an auxiliary colour set I is declared as auxiliary colour set for the places A-E to indicate the colour set for the legal annotations.

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 annotation layers on top of each other, it is possible to maintain several versions of a CP-net using one basic CP-net, and thereby to use the same basic CP-net for various purposes by adding or combining annotation layers. The modeller can for example create one annotation layer containing annotations for performance analysis, another annotation layer for visualisation using message sequence charts, and yet another annotation layer for communication with external processes which may even access information in the underlying annotation layers. By using separate annotation layers for different uses, it will be easy in a tool to enable and disable each individual annotation layer when necessary.

The semantics of annotations is established by defining a translation from an annotated CP-net to another CP-net where the annotations are an integrated part of the CP-net. This new CP-net is called the *matching CP-net*. A matching CP-net for the annotated CP-net of the resource allocation system in Fig. 4.2(a) is contained in Fig. 4.2(b). We will not discuss the translation here but refer to the paper for further details. The relationship between the behaviour of the original CP-net and the behaviour of the matching CP-net is discussed in the paper using occurrence sequences. Informally it is required 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

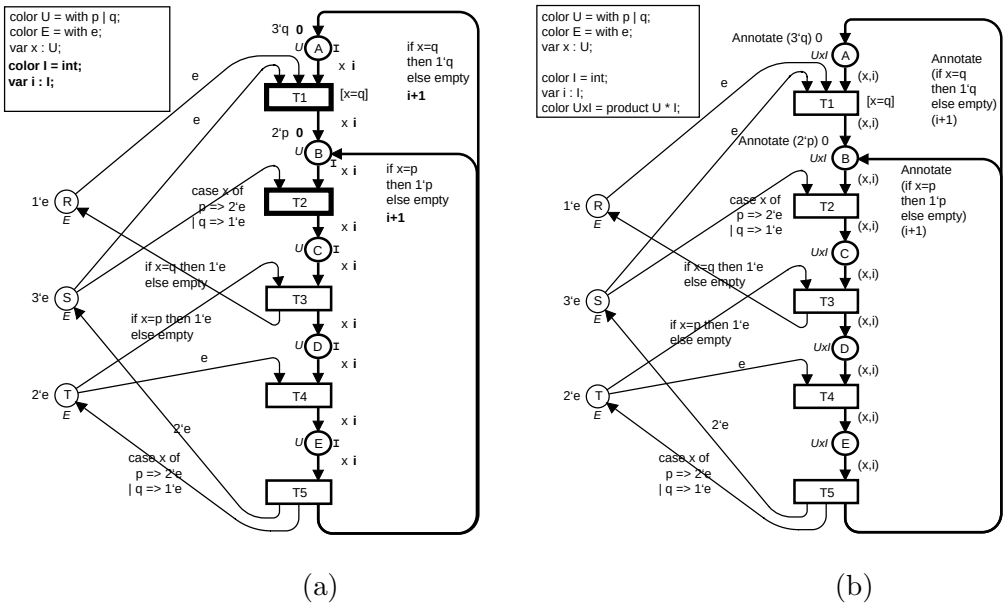


Figure 4.2: Resource allocation system: (a) An annotated CP-net with a cycle counter in an annotation layer. (b) A matching CP-net generated from the annotated CP-net in (a).

the underlying CP-net it must be possible to find at least one corresponding occurrence sequence in the matching CP-net.

Apart from defining annotations formally, the paper also gives concrete examples of how annotations can be used in practice. Support for creating annotations have not been implemented yet. However, from the examples considered manually, the proposal looks promising by separating auxiliary information from the CP-net in a very clear and clean way. By implementing the proposal in a tool and by using it for creating larger annotated CP-nets, experiences will probably indicate directions for future improvements.

In the following we summarise the advantages of annotations as proposed in the paper. One of the major advantages is that, annotations are defined so that they are ensured to affect the behaviour of the original CP-net in a very limited and predictable way. This guarantees that the overall behaviour of a CP-net is not accidentally changed when adding annotations. Therefore the behaviour from the original CP-net is preserved. Another advantage is that the semantics of annotations is defined via a translation from a CP-net and an annotation layer to another (matching) CP-net. In this way no new semantics is introduced to CP-nets. From the current definition of annotations, the translation can be conducted fully automatically. This means that the modeller do not need to know the details of the translation to be able to use annotations in practice. Defining annotations in annotation layers stresses the fact that annotations are easy to separate from the CP-net itself, and the fact that it is easy to add several different annotation layers on top of each other.

The current proposal for annotating CP-net is by no means perfect. The

current major disadvantage is the lack of tool support for annotations. In addition, the current restrictions on annotations are quite strong and have primarily been made to keep the notation as simple and intuitive as possible. Only by applying annotations on industrial sized examples we will be able to determine if the current proposal with the strong restrictions and limitations on annotations will be useful in practice.

4.3 Related Work

The further development of PT-nets to high-level Petri nets has made compact representations of systems possible. Adding information or data to tokens was the key-element in obtaining this compact representation. Annotations also heavily exploits colours in the process of adding auxiliary information to tokens. From an abstract point of view we can consider annotations as a way to add additional information to tokens – and thereby we are able to distinguish annotated tokens even though some tokens may have the same (underlying) colour. Therefore, there is a close relationship between adding colours to tokens and adding annotations to tokens.

Extending CP-nets with channels for synchronous communication is considered in [15]. The motivation for this extension came from practical use of CP-nets where it is often required to add extra places and transition to model synchronous communication. This motivated the need for explicit support for synchronous channels in CP-nets. The motivation from practical use was also one of the reasons for developing annotations to remedy laborious work with adding auxiliary information to a CP-net. The paper [15] also shows how a CP-net with channels can be transformed into a behaviourally equivalent CP-net. Like for a matching CP-net obtained from translating the underlying CP-net and the corresponding annotation layer, the behaviourally equivalent CP-net, obtained from translating a CP-net with channels, is not intended to be generated when using channels in practice. Instead, the translation is used only to define the semantics of synchronous channels and in the process of formally deriving properties of CP-nets with channels. Synchronous channels are currently not part of the distributed version of Design/CPN, but are included in the tool Renew [89] supporting reference nets [99]. Reference nets are essentially an extended version of CP-nets where tokens can be reference nets themselves. This means that tokens are not only static information, but may have dynamic behaviour and change its own value (or state). Communication between reference nets (in tokens) is conducted using synchronous channels.

Parameterised CP-nets have been examined in [19, 69]. The fundamental idea of parameterisation is to represent only the part common for all objects in a family and characterise holes of interest (parameters) which can be filled in later. In other words, when a part of a model can be used in several contexts while only modifying certain parts, these parts can be declared as parameters. The parameters can then be bound to different entities for each different use. In the papers [19, 69], different kinds of parameterisation is proposed, such as type, expression, and net parameters.

The lack of behaviour-respecting abstractions in hierarchical CP-nets has been investigated by Lakos in [50, 51]. His main purpose is not to support annotations, but rather to make it easier to prove properties about modelled systems. He proposes introducing so-called *abstraction morphisms* between CP-nets to guarantee that the structural abstraction, represented by e.g. a substitution transition, will be respected in the behaviour of the refined CP-net. As part of this proposal he defines so-called type (or colour)-refinements used to refine colours from a CP-net when used in a refined net. In other words, a *type refinement* incorporates additional information in the tokens of the refined CP-net than in the abstract CP-net. As part of the abstraction morphism, this additional information is removed when projected onto tokens in the abstract CP-net. Even though Lakos' intention is different from ours, his research stresses the fact that it is indeed necessary to be able to maintain several slightly different CP-nets of a system – and that it is important to know exactly what are the differences between these CP-nets.

In object-oriented programming languages like e.g. BETA [68], classification hierarchies are used for classifying objects which relate to each other with respect to certain characteristics. For example, consider an animal. A cat and and a bird are different animals, and can therefore themselves be considered as individual subclasses of the class animal. All animals have a mechanism which makes them able to move, e.g. cats have feet while birds have wings. In BETA, virtuality makes it possible to specify that an attribute may be specialised in a subclass – but still be manipulated at an abstract level. For example, the move-mechanism for animals is specialised to feet for cats and to wings for birds. When animals including both cats and birds are asked to move, the cat-animal will use its specialised behaviour, i.e. its feet, to move while the bird-animal will use its wings. This is somewhat similar to the way annotations extend tokens with auxiliary information. From the point of view of the underlying CP-net, a token is represented without the additional behaviour of the annotation layer, while in the matching CP-net the behaviour is extended as described by the annotation layer.

Chapter 5

Conclusions and Future Work

This chapter concludes the work discussed in this dissertation and presents directions of future work. Section 5.1 contains a brief summary of the main contributions of this dissertation. Section 5.2 presents some directions of future work.

5.1 Summary of Contributions

The focus of the research has been on providing better facilities for practical use of CP-nets. This includes facilities for creating domain-specific GUIs for CPN simulators using the tool Design/CPN, improved facilities for monitoring simulations and for annotating CP-nets. The main contributions of the work are summed up below.

- The development of tool facilities for creating web-based domain-specific GUIs for CPN simulators using Design/CPN. Creation of a domain-specific GUI makes it possible to hide a CP-net and the corresponding simulator behind a customised GUI. The developed method provides a fast way of creating a domain-specific GUI for a CPN simulator.
- A simulator for influence nets has been implemented and validated using CP-nets and Design/CPN. The simulator is created by translating an influence net to a CP-net which is then simulated using the Design/CPN simulator. The simulator is extracted from the Design/CPN tool and hidden behind an automatically generated web-based GUI which is used for controlling the simulator and for displaying simulation results using domain-specific concepts. The usability of the method has been established via evaluations with participants from the US military.
- A monitoring framework has been developed. The purpose is to separate from models the code defining how to extract and process data from simulations. Essentially, a monitor specifies when it should be activated during a simulation, how data should be extracted, and what to do with the data. Monitors can be either customised or predefined to e.g. save data in files, control stop criteria of the simulator, update message sequence charts, or send data to external processes.

- Separation of auxiliary information from CP-nets is proposed by introducing annotations and annotation layers in CP-nets. Annotations make it possible to associate auxiliary information with tokens without modifying colour sets of a CP-net. Annotation layers make it easy to maintain several different sets of annotations when using a CP-net for different purposes – instead of maintaining several slightly different copies of the same basic CP-net. In addition, the overall behaviour of the CP-net is preserved when adding annotation to a CP-net.

5.2 Future Work

The motivation and objective for the work presented in this dissertation has, as stated in Sect. 1.2, been to provide better facilities for conducting practical use of CP-nets. From the contributions described in Sect. 5.1 we claim that this has been achieved. Facilities have been developed and practical projects have shown the usability of these facilities. However, the work presented here can be improved in many ways. In this section we discuss a few ideas and directions for future work.

5.2.1 Web-Based Interfaces and Influence Nets

Currently work is conducted at George Mason University to change the influence net tool to make it possible to specify timing parameters within the influence net tool instead of specifying these parameters via the web browser. By using the influence net CPN simulator as an integrated component in the influence net tool, both input and output can be managed directly from the GUI of the influence net tool. In other words, the influence net tool itself will completely hide the fact that a CPN simulator is used internally for simulating influence nets with timing information. For some users this approach may be more appealing than having to use two different tools for creating and analysing influence nets.

Due to the interest from the US military in using CPN simulators for influence nets, it may also be worth while to investigate how influence net CPN simulators can be integrated with other simulation models. The main reason why this may be relevant is that the US military put great effort into developing a high-level architecture (HLA) [93]. HLA supports the combination of several simulation models into so-called federations of simulation models. In other words, the architecture makes it easy to create a distributed environment for running several interacting simulation models.

5.2.2 Monitors

In the paper on monitors we claim that the monitoring framework can probably be used for any discrete-event system formalism. This claim has to be investigated further in the future by investigating and evaluating other discrete-event system tools.

As mentioned previously, the monitoring framework has been developed as a generalisation of the data collection facilities in the Design/CPN Performance Tool. The Design/CPN Performance Tool has been used in several projects, and was integrated in version 4.0 of Design/CPN released in September, 1999. In contrast, only a prototype of the monitoring framework has been implemented and applied to a few CP-nets. Future work should focus on completing the implementation of the monitoring framework. This includes extending the GUI of CPN Tools [23] which is the tool going to replace Design/CPN in the near future. In this tool the design of the GUI for the monitoring framework should use the new advanced user interaction techniques as described in [6, 5].

Future work could also continue our undocumented work on *batch monitors*. The monitors described in the monitoring framework are primarily intended to be applied during a single simulation. When using a model for e.g. performance analysis it is often necessary to run several simulations – possibly with slightly different parameters. To support monitoring during such batches of simulations we have considered how batch monitors could be designed. We believe that there is a great potential for batch monitors, and it should definitely be investigated further in the future.

5.2.3 Annotations

There are many issues regarding annotations that can be addressed in the future. First of all, the ideas on annotations presented here have not yet been used in practice due to lack of tool-support. It is of high priority to develop a prototype of a tool supporting annotations. Only by using annotations in practice on concrete CP-nets, it is possible to determine if annotations will make it easier for the modeller to maintain auxiliary information within a CP-net. The prototype of a GUI supporting annotations should be implemented in CPN Tools where the design of the GUI could use the new advanced user interaction techniques of CPN Tools.

Further research is required on annotating arc expressions on arcs that evaluate to multi-sets instead of single colours. The current proposal where all colours in the multi-set will get the same annotation may be too restrictive in practice. However, if the notation of annotations is too complex, then users may be reluctant with learning how to use annotations. Therefore, the limitations have been made to make the annotation notation as simple as possible – and thereby to make it easier to design tools with user-friendly support for annotations.

Guards are not considered in the proposal for annotations. The main reason is that guards are often added to restrict the enabling of transitions. However, guards can also be used for binding free variables. A future proposal for annotations should consider the ability of binding free auxiliary variables in annotation layers by means of guards.

Our proposal only considers how to add annotations to existing initialisation and arc expressions, and thereby only considers how to annotate existing tokens. However, it might be useful also to be able to add auxiliary net structure to

the annotation layers. As an example, a place could be added only to the annotation layer with a token holding a counter with the number of occurrences of a transition. Allowing such additional net structure in 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 annotations.

Another issue that should be addressed in the future is whether or not the translation from a CP-net and the corresponding annotation layer to the matching CP-net should be conducted in a tool. The author of this dissertation currently believes that the translation should be conducted implicitly by modifying the simulator in the tool. The simulator should be modified so that the annotations are handled explicitly by the simulation tool instead of being integrated into the CP-net. In other words, the translation should only be used for defining the semantics of annotations and should not be applied to every annotated CP-net. However, this should be investigated in the future.

Finally, the similarities between the monitoring framework and annotations could be considered. Both monitors and annotations aim at separating code and information from a CP-net. Annotation layers have the advantage compared to monitors, that there is a tight coupling between the annotations and the graphics of the CP-net while still easily being able to disable annotations when needed. Therefore, it should be investigated how these advantages can be used for integrating monitors into annotation layers.

Part II

Papers

Chapter 6

Web-Based Interfaces for Simulators of Coloured Petri Net Models

The paper presented in this chapter was first published in the proceedings of the Workshop on the Practical Use of High-Level Petri Nets 2000. Later the paper was accepted for publication in the *International Journal on Software Tools for Technology Transfer (STTT)*, 2001.

[53] B. Lindstrøm. Web-Based Interfaces for Simulators of Coloured Petri Net Models. In K. Jensen, editor, *Workshop on the Practical Use of High-Level Petri Nets, DAIMI PB–547*, pages 15–33. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/-pn2000/proceedings.

[54] B. Lindstrøm. Web-Based Interfaces for Simulators of Coloured Petri Net Models. *International Journal on Software Tools for Technology Transfer*, 3(4):405–416, September 2001.

Except for minor typographical changes the content of this chapter is equal to the STTT paper in [54].

Web-Based Interfaces for Simulation of Coloured Petri Net Models

Bo Lindstrøm*

Abstract

This paper describes an approach which allows users without knowledge of coloured Petri nets to control the simulation of coloured Petri net models and interpret the results obtained from simulations via web-based interfaces. We describe the architecture design of facilities in a simulation tool for making it possible to simulate a coloured Petri net model via a web-based interface. The approach is based on giving the modeller the ability to easily create a CGI script containing the entire simulator. The initial conditions of the simulator for running several simulations can then be specified via a so-called HTML form on a web page. The fact that the initial conditions of a simulation can be specified via a domain specific and user-relevant form, gives users without knowledge of coloured Petri nets and the actual simulation tool the ability to use the pre-constructed simulators for specific analysis purposes. The paper also illustrates that integrating so-called batch scripts into a CGI script has some advantages. Batch scripts give the user the ability to run several simulations after having specified initial conditions for each simulation. Thus we will be able to first specify input via a web page and then based on that input run several simulations. As a representative example we show how to implement the approach in the Design/CPN tool.
Key words: Coloured Petri nets – Batch simulations – CGI scripts – Design/CPN – HTML forms – Web interfaces.

6.1 Introduction

There are many situations in which we can use models of systems to help in making decisions about the operation of the modelled system. In many cases discrete event models are the most appropriate computational engines for these decision support tools. Petri nets, in general, and coloured Petri nets [44] (CPNs or CP-nets) in particular, are general modelling languages for creating discrete event system models. Petri nets support both analysis of the logical properties of systems and simulation, so that logical properties and behaviour of systems can be examined [45]. Petri nets can also be used to investigate the performance of systems. While Petri net tools, such as Design/CPN [27],

*Department of Computer Science, University of Aarhus, Denmark, E-mail: blind@daimi.au.dk

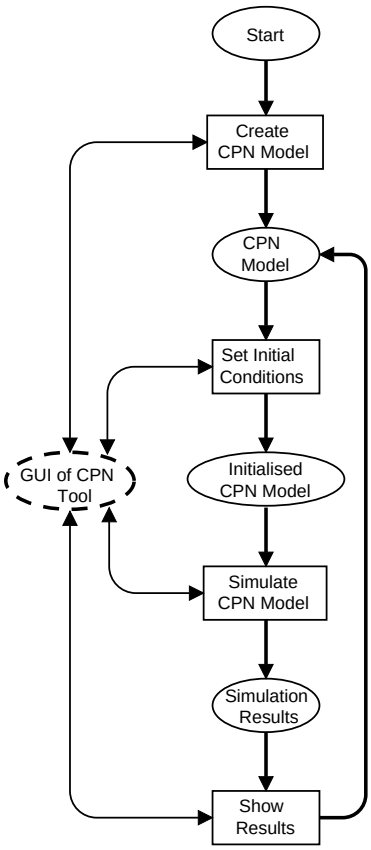


Figure 6.1: Original approach for creating and simulating CPN models.

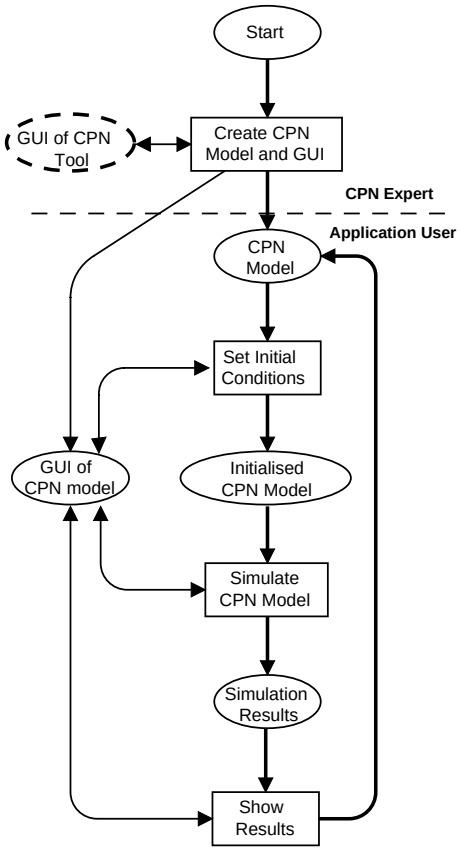


Figure 6.2: New approach for simulating CPN models via a web browser.

offer powerful capabilities for verification [47] and performance analysis [58] of models, their complexity and the need for understanding Petri net theory requires specially trained personnel.

The motivation behind this paper has been to make Petri net technology available for application users who are not experienced with Petri nets. The development of CP-nets and their tools has progressed to an industrial strength modelling language that retains the theoretical foundation of Petri net theory. However, not much focus has been on integrating simulation models into real applications.

Until now, the same graphical user interface (GUI) is often used for all activities involved in creating, simulating and analysing CPN models. Figure 6.1 illustrates this approach. First, the GUI (the dashed place) is used to create the CPN model. Then the same GUI is used for simulation activities, such as setting the initial state or initial conditions of the model, and afterwards, it is used for simulating the CPN model. Finally, the simulation output produced during simulations is often also displayed using the same GUI.

The architecture envisioned in this paper is to leave the creation of the CPN models to CPN experts, and let the application users simulate the models

using another domain specific interface (see Fig. 6.2). First of all, the CPN expert creates a CPN model together with a suitable GUI for applying the CPN model. Creating the CPN model and the GUI tailored to the CPN model allows application users without knowledge of the simulation tool to simulate the CPN model over a range of conditions (initial markings). Application users use the specially tailored GUI to set the initial state/conditions of the CPN model and then to run the simulation. Finally, the domain tailored or domain specific GUI may be used to display the results of the simulation.

The architecture described above using domain tailored GUIs was expanded to provide the application user with remote access to the simulator of a CPN model using web technology such as so-called Common Gateway Interface scripts (CGI scripts) [35]. Thus, the application user has the advantage of having access to a rigorous discrete event system model of a complex distributed or concurrent system over a network using any web browser. The application user controls the executable model through a *Hyper Text Markup Language* (HTML) [102] form that provides inputs to the CPN model. The output will possibly be returned to the web browser of the application user. In this manner, the application user can perform analysis of the CPN model of the system without needing to understand or even see either the CPN models and CPN formalism or the user interface of the CPN tool. Thus, the application user simply provides input via a form in a HTML document which is tailored to the specific CPN model and uses terminology and concepts from the application domain, while the actual CPN model and the modelling tool are never seen by the application user. The remaining sections of this paper provide a detailed description of the design, implementation, and use of the approach outlined above. The Design/CPN tool was used as a basis for the implementation of the approach.

The paper is organised as follows. Section 6.2 presents a simple but realistic example which illustrates the approach for accessing simulators of discrete event systems via a web browser. This example will be used as a running example throughout the rest of the paper. Section 6.3 contains a general discussion of the design considerations related to developing the approach. Section 6.4 describes how to realise the design in the Design/CPN tool. Section 6.5 illustrates the amount of code a CPN expert using Design/CPN needs to write to apply the approach to a specific CPN model. Finally, section 6.6 concludes the paper and gives suggestions for future work.

6.2 Example: Backup Company

This section gives an example of a scenario, where a web-based interface for simulating CPN models can be very useful. We will use the example presented in this section for illustration purposes in the rest of the paper. We will not describe the CPN model itself because it is not necessary to understand the details of the CPN model in order to understand the approach of simulating a CPN model via a web browser. In later sections we will discuss how to modify the environment of a CPN model in order to allow it to be simulated via a web browser. We stress that the company presented in the example is fictive, but

the example is simple and illustrates the possibilities of the approach.

A company sells backup devices from a web site. When a customer needs to decide which backup device to buy for a complex network environment, it is necessary to take several factors into account. The customer needs to consider his existing system in order to obtain the most suitable backup system. In particular he needs to consider the following factors.

- What is the network bandwidth on the local area network?
- How many machines exist of each type: servers and workstations?
- How much disk capacity is present?
- Which is most important: price or performance?

Today many of the customers call the company and ask what components they should buy for their particular system. The company has created a CPN model that can provide a specification of the needed backup system given system requirements like the ones stated above. The CPN model helps the employees answering the phone calls to answer the questions of the customers.

It takes a lot of time for the company to answer questions from customers who ask what to buy for their specific system. Therefore, the company wants to provide their potential customers with a tool that can help decide which components to buy. An obvious solution would be to simply give the customers access to the CPN model from a web page. In this way both answering technical questions about what to buy and placing an actual order can be handled via the Internet. This means that customers do not have to call the company to figure out what to buy, thus reducing the service costs for the company.

After having applied the approach described in this paper to the CPN model, it is possible to simulate the CPN model via a web browser. A customer who wants to buy some backup devices accesses the web page containing an HTML form, like the one in Fig. 6.3, which is the interface to a simulator of the CPN model. The customer types in the data of the existing system together with the requirements of the new system into the form.

The system replies after having simulated the CPN model based on the input given in the form (see Fig. 6.4). Different kinds of output are produced. A textual description of the necessary backup components is given, and the expected performance of alternative components is illustrated using graphs.

The CPN model used in this example is a customised model which can simulate any backup device the company sells. The model is fixed to a specific backup device by using different initial markings.

To determine which backup device is the most suitable for a customer, the CPN model is simulated several times with different initial markings – once for each backup device that the company produces. Based on the results obtained from the simulations, the most suitable ones are selected and displayed for the customer.

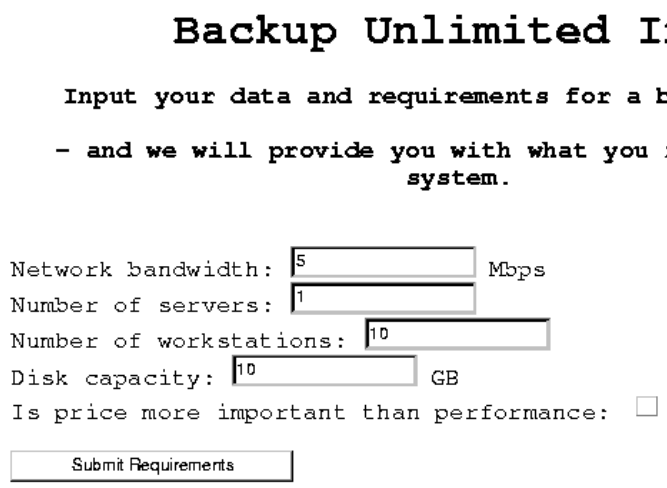


Figure 6.3: HTML form as interface to the CPN model.

6.3 Design of the Web-Based Approach

The general description of the approach for creating web-based interfaces for simulating CPN models introduced in Sect. 6.1 will be considered further in this section. Section 6.4 will focus on how to implement the design in the Design/CPN tool.

Several different options exist for executing and controlling a program via a web browser. In this paper we describe an approach for controlling simulations using so-called *Forms* and *Common Gateway Interface scripts* (CGI scripts) [35], which are both well-known Internet techniques.

Figure 6.5 illustrates the setup for using CGI scripts, while the message sequence chart in Fig. 6.6 illustrates what happens when a CGI script is activated by submitting a form from a web browser. Assume that a hypertext document (`backup_form.html`) is located on Computer B which is a web server (HTTP server B). The document could, e.g. be an HTML document producing a form like the one in Fig. 6.3. The application user using the web browser at Computer A downloads the HTML document. The application user then fills out and submits the form. When the application user submits the form, a link (URL) to a file on Computer C that holds the CGI script (`backup.cgi`) will be followed. (The CGI script may also be placed on the same computer as the HTML form.) This link is a "normal" HTTP link, but the file on the web server on Computer B is stored in such a way that the web server on Computer C can tell that the file contains a CGI script that is to be executed, rather than a document that is to be sent to the client as usual. The web server then executes the CGI script which can read the input that the user typed into the form. Based on the input the CGI script dynamically generates an HTML document. The HTML document is sent to the client while it is being generated as a stream. The web browser on the client computer displays the document while receiving

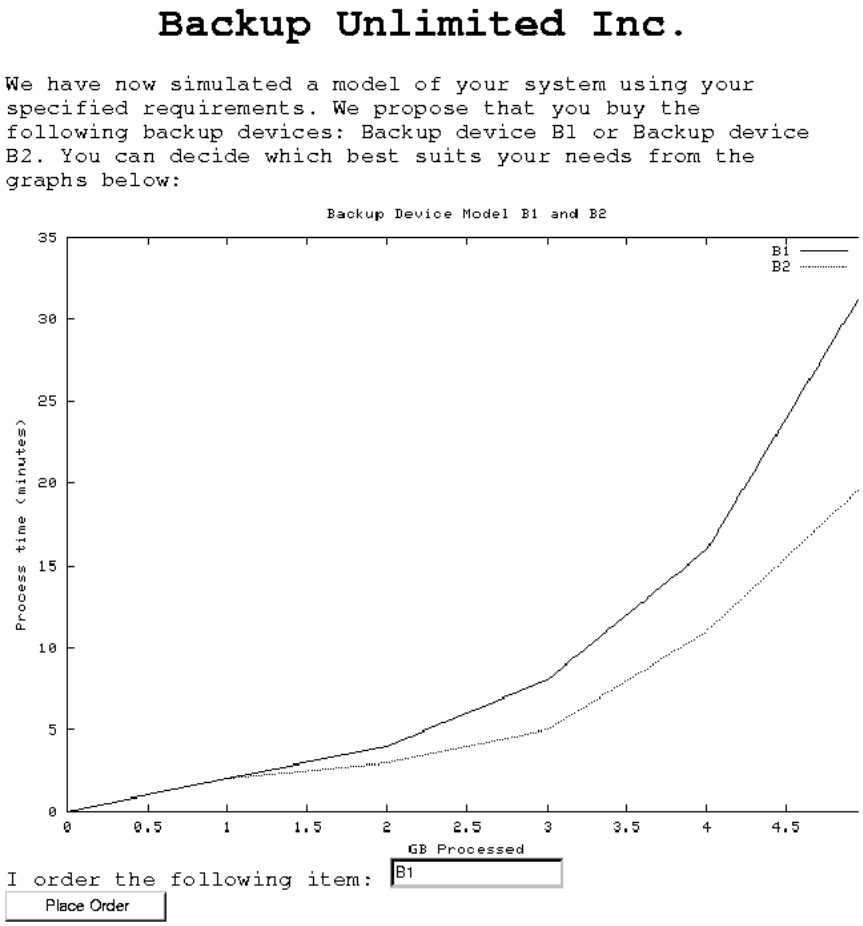


Figure 6.4: An HTML document containing the results of simulating the CPN model.

the stream of HTML code from the web server, as it would display any other HTML document. The HTML document being received could, e.g. be like the one in Fig. 6.4.

Forms are very useful for specifying textual input via a web browser. Furthermore, they provide a simple mechanism for submitting the contents of the form to a CGI script. The HTML code for specifying a form is also very simple. Figure 6.7 shows the HTML code used to display the HTML document in Fig. 6.3. The URL `www.daimi.au.dk/cgi-sim/cpn.cgi` in line 3 identifies the CGI script to be activated when the user submits the form. The button for submitting the form is created using line 15 in Fig. 6.7. Lines 9 – 14 specify that five fields for input should be created. The input fields are named uniquely in the form using names (`bandwidth`, `servers`, `clients`, `disk`, and `price`). These names are used by the CGI script to access the values of the fields when the form is submitted.

Note that only a few lines of HTML code are necessary to create a form,

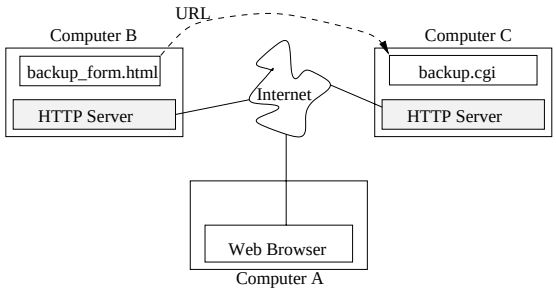


Figure 6.5: A web browser and two web servers.

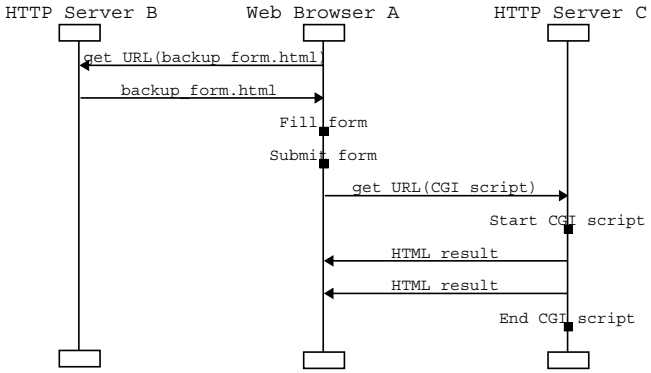


Figure 6.6: A web browser requests a CGI script to be executed.

thus most people who can read CPN models and do some basic programming should be able to learn to write such HTML code without too much difficulty.

The program or CGI script that reads the information submitted in the form and that processes the information is more complex. Specialised scripts are required to handle the incoming form information. CGI scripts may be written in scripting languages like Perl [109] or in programming languages like C and Standard ML (SML) [83]. In general, a CGI script can be considered as an executable program that can be executed on a web server by request from a web browser.

To be able to simulate CPN models using CGI scripts in a controllable manner there are some requirements of the Petri net tool. First of all, the Petri net tool should be able to start a simulation of a specific CPN model without requiring a user to interact with the tool-specific GUI. For example, it should not be necessary to use dialog boxes in the Petri net tool to start a simulation – it should be possible to automate everything in a script. The tool should allow users to write user-defined functions, e.g. for retrieving input, saving files, and printing to standard output. The reason for these requirements is that because a CGI script is invoked via a form, the CGI script must be able to control everything related to reading input from the form, setting the initial state, starting the simulation, and producing results.

To make the CGI script as user-configurable as possible with respect to controlling a simulation, the CGI script must be able to control the simulation

```
<HTML><BODY>                                                              1
<FORM method=GET
     action="http://www.daimi.au.dk/cgi-sim/cpn.cgi">                     3
 <CENTER>
   <H1>Backup Unlimited Inc.</H1>                                         5
   <H3>Input your data and requirements for a backup system</H3>
   <H3>- and we will provide you with what you need for your system.</H3>  7
 </CENTER>
 Network bandwidth: <INPUT size=15 type=text name=bandwidth value=5>Mbps<BR> 9
 Number of servers: <INPUT size=15 type=text name=servers value=1><BR>
 Number of workstations: <INPUT size=15 type=text name=clients value=10><BR>11
 Disk capacity: <INPUT size=15 type=text name=disk value=10>GB<BR>
 Is price more important than performance:                               13
     <INPUT size=15 type=checkbox name=price><BR><BR>
 <INPUT type=submit value="   Submit Requirements  ">                    15
</FORM></BODY>
</HTML>                                                                   17
```

Figure 6.7: HTML code for creating a form.

tool in the following way: when the CGI script is executed from a web browser, it should automatically execute a sequence of commands. In the following we will refer to this sequence of commands as a batch script [57]. A batch script can be considered as a simulation control script with the purpose of specifying exactly what the CGI script is intended to do – including when to start a simulation. In this context, a CGI script will be defined as an executable CPN simulator together with a batch script which defines what happens when the CGI script is executed.

To give the user the largest possible freedom for defining CGI scripts the batch script needs to be able to be specified by the user. In the context of CGI scripts the batch script typically has a certain structure which is likely to include the following actions:

i. Retrieve parameters from the form (see Fig. 6.7 for example of HTML code for a form).

ii. While there are more simulations to run do the following:

   (a) Calculate markings to be used to initialise the state of the CPN model.

   (b) Initialise the state of the simulator.

   (c) Run simulation – and collect data.

   (d) Save results and/or send HTML code to the client web browser.

iii. Quit the CGI script.

First of all, the batch script needs to retrieve the parameters that are sent to the CGI script from the form. After having extracted the data from the input fields in the form, the batch script needs to specify how to run the simulation(s).

```
<HTML><BODY>                                                                    1
<FORM method=GET action="http://www.daimi.au.dk/cgi-sim/place_order.cgi">
 <CENTER>                                                                       3
  <H1>Backup Unlimited Inc.</H1>
 </CENTER>                                                                      5
 We have now simulated a model of your system using your specified
 requirements. We propose that you buy the following backup devices:            7
 Backup device B1 or Backup device B2. You can decide which best
 suits your needs from the graphs below:                                        9
 <BR>
 <CENTER>                                                                      11
  <IMG SRC="http://www.daimi.au.dk/result.graph.png">
 </CENTER>                                                                     13
 I order the following item: <input size=15 type=text name=no value=B1><BR>
 <INPUT type=submit value="   Place Order  "><br><br>                          15
</FORM></BODY>
</HTML>                                                                        17
```

Figure 6.8: HTML code for creating the form in Fig. 6.4.

The batch script often starts by calculating the markings for the initial state. The calculation is likely to use input parameters from the form or results from previous simulations. Now the state of the simulator is ready for starting the simulation.

While the simulation runs, data is often collected and saved in files for later processing. Therefore, the model often needs to be instrumented to collect the needed data. When a simulation has finished, the results collected during simulations may be sent directly to the web browser (by printing HTML code to standard output). In addition, results from the simulation may be saved to files for later post-processing. Finally, the batch script may decide that more simulations are to be performed. Then the batch script may continue by restarting the script, otherwise the CGI script terminates.

Simulation tools that support converting a simulator into a CGI script may also include some auxiliary and high-level facilities. Examples of such high-level facilities are facilities for creating graphs to be saved in files, and facilities for printing HTML code for referring to graphs in the HTML document being generated by the CGI script. This makes it possible for the web browser to download the image containing the graph while displaying the HTML document. Such an image `result.graph.png` was included in the HTML document in Fig. 6.4 in Sect. 6.2 using HTML code like line 12 in Fig. 6.8.

If the simulation tool does not contain the needed post-processing facilities itself, it may possibly use external programs for post-processing of the data. There are only two requirements of the post-processing tool to allow using it from the CGI script. The first one is that the tool should support being executed from the command line using a script containing all the needed commands to create the graph. Secondly, the tool should be able to save the output to files. Gnuplot [32] is an example of a tool that can be used for generating plots and graphs, and it supports command line scripts, too.

Given a CPN tool that fulfills the above mentioned requirements by allowing

to create batch scripts and then be remotely controlled, it is possible to create CGI scripts that fulfil many needs for simulation of CPN models from web browsers. In total the CPN expert needs to create two different files: an HTML form (like Fig. 6.7) to be used for specifying input to the CGI script, and the batch script for retrieving input from the form, running the simulation, and for producing results to be displayed at the web browser.

In the approach described in this paper, all input from the application user should be ready before submitting the form. In this situation it is not possible to control the simulation after it is started – the simulation will be non-interactive but results can be shown gradually as they are generated. In Sect. 6.6 we discuss future work which addresses how to obtain interactive simulations within the approach.

6.4 Implementation in Design/CPN

Design/CPN [27] is a widely used tool supporting editing, simulation and verification of CPN models. In this paper, Design/CPN is used to prove the usefulness of the CGI concept for simulating CPN models via a web browser. In this section we describe some of the design considerations and facilities that are implementation specific for Design/CPN.

According to the description in Sect. 6.3, a CGI script is nothing more than a program that can be executed from a web page and then dynamically produces a new web page. A few modifications are made to Design/CPN in order to make it possible to turn the simulator of Design/CPN into a CGI script, or rather to drive the Design/CPN simulator from a CGI script. To understand why these changes need to be done, we need to describe some of the architecture of the Design/CPN tool.

Design/CPN is divided into two parts: one part implementing the GUI, and another part containing a simulation engine for simulating a CPN model. When a CPN expert has created a CPN model (see Fig. 6.9), simulator code can be generated containing the simulation engine and some model dependent code. This code contains everything needed to simulate that specific CPN model. Figure 6.9 differs a little from Fig. 6.1 in that a simulator is generated from the CPN model before being able to perform the actual simulation. However, when the simulator has been generated once and for all, it may be used to run any number of simulations.

In the context of CGI scripts only the simulator of Design/CPN is of interest. The reason is that it contains the entire executable simulator which is generated from the CPN model created in the editor of Design/CPN. Thus, the Design/CPN simulator is the only part of Design/CPN that needs to be used when creating a CGI script.

6.4.1 Disabling the GUI of Design/CPN

Due to the fact that the simulator and the graphical user interface (GUI) of Design/CPN communicate with each other, the Standard ML (SML) [83] functions contained in the simulator for updating the GUI of Design/CPN should

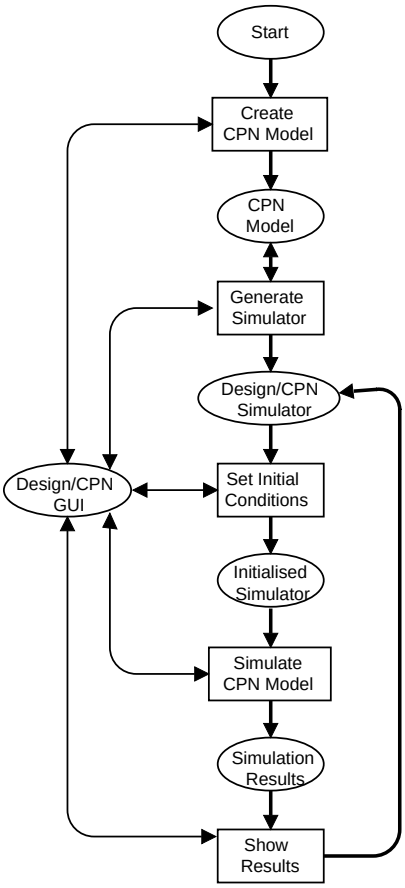


Figure 6.9: Design/CPN approach for creating and simulating CPN models.

be modified in order to not update the GUI while simulating. The reason for this modification is that the GUI of Design/CPN is not present in a CGI script. Only the code constituting the simulator and user-defined functions is contained in a CGI script. Remember that a CGI script is invoked from a form in a browser, while the CGI script itself is running on the web server. Therefore, the web page shown by the web browser can be considered to be the GUI of the CGI script.

6.4.2 Creating CGI Scripts

Another modification of Design/CPN makes it possible to save a CGI script in an executable file. The executable file will contain the entire simulator code for the CPN model and a user-defined function. The simulator code is model dependent SML code which is automatically generated by Design/CPN. This code makes it possible to simulate the CPN model. The user-defined function will be a batch script, as described in Sect. 6.3. Batch scripts in Design/CPN are written in SML.

An example of a batch script can be found in Fig. 6.10. The function `getValOfField` reads the value that the user has typed into the form in the field

```
fun batch_script (_, _) =
    let fun parse_form_input () =
            {diskField=getValOfField "disk",...};
        fun cycle_script () =
            (calculate_initial_marking();
             init_state(); (* Initialise state of the simulator *)
             simulate(); (* Run the simulation *)
             save_results();
             if not_finished() then cycle_script ()
             else ());
    in
        (parse_form_input ();
         cycle_script ())
    end
```

Figure 6.10: A simple batch script.

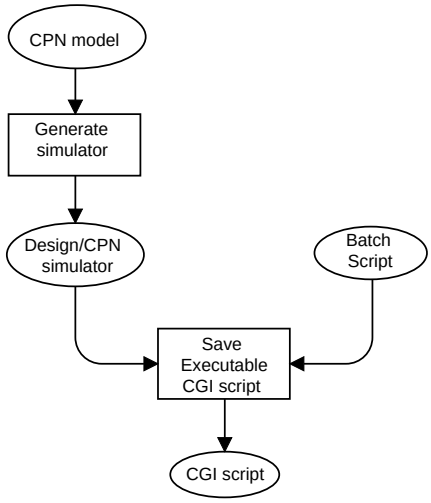


Figure 6.11: Creating a CGI script from Design/CPN.

named by `disk` which is the name of the field with the title "Disk Capacity" in Fig. 6.3. The function `parse_form_input` is simplified here to only read the field `disk`. The user may want to do some calculations (`calculate_initial_marking`) before initialising the state of the simulator by means of the function `init_state`. The simulation is started using the function `simulate`. The function `save_results` saves results in files and/or sends them to the web browser.

When the user has finished creating the batch script to be included in the CGI script, the actual CGI script can be generated. The CGI script is generated and saved in a file by simply invoking a SML function. The overall approach for creating a CGI script using Design/CPN is illustrated in Fig. 6.11. First the model is created and the simulator is generated using Design/CPN. Then a batch script is created by the CPN expert to control the actions of the CGI script. Finally, the CGI script containing the batch script and the Design/CPN simulator is generated and saved in a file.

```
fun print_header n =
   (print ("<B><FONT SIZE=+2>Results from running "^
           (Int.toString n)^
           " simulations</FONT></B>"));
```

Figure 6.12: Print HTML code to a web browser.

6.4.3 High-level Functions

Due to the fact that SML is the language used for specifying CGI scripts which contain the Design/CPN simulator, or in particular that the batch script contained in the CGI script is written in SML, the CPN expert can use the full power of the language SML for getting input and producing output from the CGI script. This section describes some auxiliary and high-level functions that may be useful when creating batch scripts to be used in CGI scripts. In particular we will focus on functions for reading input from HTML forms and generating HTML code as simulation output.

We will now describe how to read the parameters entered in the browser from the CGI script. When discussing the general structure of a batch script in Fig. 6.10, we introduced the SML function `getValOfField`. This function is very useful when reading a value that an application user has input in a form. The function can be used to read any input field in a form, thus it is very general. The only parameter given to the function is the name of the field in the form. The function retrieves the input sent from the form and returns the value contained in the field.

When we want to produce an HTML document as output from running a CGI script, the CGI script needs to print some HTML code to standard output. SML provides the function `print` for printing to standard output. By printing HTML code to standard output it is possible to create complex web pages. The HTML document may even include embedded graphics, so-called Java scripts [95], and Java applets which allow programs to be executed directly on the client machine holding the web browser. Figure 6.12 illustrates a simple SML function which prints some simple HTML code.

It may also be useful to save HTML documents as files. This is particularly useful if several data files and figures are generated by the CGI script. In this way it is possible to divide the results of running the CGI script over several HTML documents. The page printed directly to the browser could simply be a kind of index page for the rest of the HTML documents. In this way the results of executing the CGI script may be shown in a structured manner. SML contains several functions for creating files and directories. Thus, it is immediately possible to save several HTML documents, to be referred to from other HTML documents.

Results or raw data obtained during or after a simulation can also be saved in files. Collecting data can, e.g. be done using the Design/CPN Performance Tool described in [58]. By printing HTML code which provides an URL link to the data file (see Fig. 6.13) to the web browser, the user can download the

```
fun print_url (URL, description) =
   (print ("<A HREF=\""^URL^"\">"^description^"</A>"));

print_url ("www.daimi.au.dk/res1.txt",
          "Results from simulation");
```

Figure 6.13: Print URL to a web browser.

```
fun gen_gnuplots
        {filenamesxtitles : (string * string) list,
         title : string,
         xlabel : string,
         ylabel : string,
         dest_filename : string,
         gnuplot_path : string}
```

Figure 6.14: Interface to Gnuplot function.

result files when the CGI script (or simulation) ends. After downloading the file, the user can analyse the results produced using his favourite analysis tool. As an alternative to downloading the raw data, the CGI script can post-process the data itself.

In Sect. 6.3 we discussed generating graphics using external programs directly from the CGI script. We said that Gnuplot [32] is a tool that can be controlled from a CGI script. To make it as easy as possible to create plots using Gnuplot, and because Gnuplot is a non-commercial product, we provide a SML function for plotting graphs using Gnuplot. Figure 6.14 contains the interface of the Gnuplot SML function. The function is simply called with a list of titles and file names of the raw data files and some textual information to be included in the graph. Finally, the user also needs to specify a destination file name where the plot is supposed to be saved. This SML function implies that users not familiar with Gnuplot are also able to easily create plots using the tool. The plot in Fig. 6.4 was generated using this function.

6.5 Creating CGI Scripts in Design/CPN

In this section we give an overview of how simple it is to create CGI scripts from Design/CPN. In particular we illustrate how a typical batch script to be included in a CGI script will look.

Below we include most of the SML code for creating the CGI script used in Sect. 6.2. The purpose of including the code is to give the reader an idea of the complexity and the amount of code to be written to create CGI scripts. It is not important to understand every detail of the code. Some of the functions which are not directly associated with CGI scripts and batch scripts are not defined here. Please note that the code is rather general and can easily be modified to be used for another CPN model – or even be generated automatically.

```
fun batch_script (_, _)=
    ((* Retrieve parameters from form *)
     update_model(retrieve_input ());

     (* Run one simulation for the 20 different backup devices *)
     run_simulations (1, 20);

     (* Generate output *)
     gen_output ());
```

Figure 6.15: Batch script.

```
(* Retrieve data from the form *)
fun retrieve_input () =
     {networkField      = getValOfField "bandwidth",
      serversField      = getValOfField "servers",
      workstationsField = getValOfField "workstations",
      diskField         = getValOfField "disk",
      priceField        = getValOfField "price"}
```

Figure 6.16: Retrieve input.

Figure 6.15 contains the SML function which is to be invoked when the CGI script is executed. First the function retrieves the input that an application user has entered in the form, and then updates the CPN model with the extracted data. The function `update_model` is not included here. Then 20 simulations are executed to investigate the CPN model using the input parameters just retrieved from the form. Finally, some output is generated and sent to the web browser.

The function `retrieve_input` for retrieving the data input by an application user in the HTML form is contained in Fig. 6.16. The contents of each field is retrieved using the predefined function `getValOfField`. The function is very simple, and in the future it may be possible to generate the code for the function automatically.

When all parameters are retrieved from the form, we define how to run the simulations. Figure 6.17 describes how to configure the CPN model, run the simulations, save results, and finally decide if further simulations are to be performed.

The function `load_model_configuration_parameters` is not contained here. The purpose of this function is to load some configuration parameters into the CPN model. These parameters are supposed to specify initial markings which do not depend on the input from the form. In the context of the backup CPN model, these parameters would specify the configuration of the specific backup device to be simulated, e.g. capacity on tape, speed, etc.

After loading configuration parameters, the state of the CPN model can be initialised using the function `init_state`. To collect the needed results from the simulation, the CPN expert may have defined some functions for collecting data.

```
(* Specification of how to run simulations *)
fun run_simulations (i, (n:int)) =                                    2
    if (i <= n) then
        (load_model_configuration_parameters i;                       4
         init_state();
         createDataCollectors i;                                      6
         simulate();  (* Simulate the CPN model *)
         if (do_results_suit_user_needs ()) then                      8
             remember_model_no i
         else ();                                                     10
         run_simulations (i+1, n))
    else ();                                                          12
```

Figure 6.17: Run simulations.

We assume that this is done using a function named `createDataCollectors`. Now the CPN model is ready to be simulated. The simulation is executed using the function `simulate`. When the simulation ends we can examine the state, and observe if the results obtained suits the needs that the application user has initially requested using the form. We assume that the function `do_results_suit_user_needs` takes care of that. Finally we call the function `run_simulations` recursively for possibly running yet another simulation.

Figure 6.18 contains the function `gen_output` which produces the HTML document to be displayed at the web browser. The output includes both textual information and a graph. The graph is plotted using the function in Fig. 6.19. The graph is saved in a file with a unique file name, and appropriate HTML code referring to the graph is printed to standard output. In this way the web browser receiving the HTML code from the CGI script automatically downloads and displays the graph.

6.6 Conclusion and Future Work

In this paper we have described how a web interface to a simulator of coloured Petri net models can be designed. In particular we have illustrated how it has been done in the Design/CPN tool. The approach is based on giving CPN experts the ability to easily create a CGI script containing the entire simulator. The initial conditions of the simulator can be specified via an HTML form on a web page. The fact that the initial conditions of a simulation can be specified via a domain specific form, gives users without knowledge of either CPN or Design/CPN the ability to use pre-constructed simulators for specific analysis purposes.

The paper has also illustrated that integrating batch scripts into a CGI script has some advantages. Batch scripts give the user the ability to run several simulations after having specified input for all the simulations. Thus we will be able to first specify input via a web page and then based on the input run several simulations. The fact that the input to the CGI script can be specified in a HTML form on a web page means that the interface to the simulator can

```
fun gen_output () =
   ((* Print HTML directly to web browser *)                              2
    print "Content-type: text/html\n\n"; (* CGI-header *)
    print "<HTML><BODY BGCOLOR=#FFFFFF>";                                 4
    print ("<FORM method=GET action=\"http://"^
           "www.daimi.au.dk/cgi-sim/place_order.cgi\">");                 6
    print "<CENTER>";
    print "<H1>Backup Unlimited Inc.</H1>";                               8
    print "</CENTER>";
                                                                          10
    (* Print the results of running the simulations *)
    print ("We have now simulated a model of your system "^               12
           "using your specifiedrequirements. We propose "^
           "that you buy the following backup devices:");                 14
    print (model_alternatives (!alternatives));
    print "You can decide which best suits your needs "^                  16
          "from the graphs below:<BR>";
    print "<CENTER>";                                                     18
    (* Generate graphics using Gnuplot *)
    print_graphics (data_files());                                        20
    print "</CENTER>";
    print "I order the following item: <input size=15 "^                  22
          "type=text name=item_no><BR>";
    print "<INPUT type=submit value=\"  Place Order  \">"^                24
          "<BR><BR>";
    print "</FORM></BODY></HTML>");                                       26
```

Figure 6.18: Generate output similar to Fig. 6.4.

be domain specific and configurable. The domain specific and user-relevant graphical interface to simulators makes simulations of CPN models accessible for non-CPN experts.

Some Petri net based tools allow a similar architecture to the one presented in this paper. A tool like e.g. ExSpect [28] can be used as a COM component which makes it possible to create facilities similar to the ones described in this paper. Furthermore, ExSpect allows using so-called dashboards which is a domain specific interface for a Petri net based engine.

Future work may include investigating the ability to explore state spaces via a browser – again possibly with a domain tailored web page as graphical interface. This will also make it possible for non-experts to use the power of state spaces for answering questions by querying the state space of a CPN model. This could be obtained using the occurrence graph tool [47] of Design/CPN via a CGI script. It will be immediately possible to explore state spaces from a web page using the approach described in this paper. In Design/CPN it is just a matter of saving a CGI script after generating code for the occurrence graph tool instead of the code for the simulator.

Another interesting area is interactive simulation control via Java applets [95] embedded in HTML documents. It will be possible to implement a domain specific GUI giving interactive control of the simulator of Design/CPN using a Java applet. Java applets are small Java programs that are automatically

```
(* Create a plot using Gnuplot *)                                    2
fun print_graphics (datafilesxtitles) =
    let val unique_filename = get_unique_filename ();                 4
    in (gen_gnuplots {filenamesxtitles = datafilesxtitles,
                      title= ("Backup Device Model "^                 6
                              (model_alternatives (!alternatives))),
                      xlabel = "GB Processed",                        8
                      ylabel = "Process time (minutes)",
                      dest_filename = unique_filename,                10
                      gnuplot_path = "/usr/local/bin/gnuplot"};
        print ("<IMG SRC=\"http://www.daimi.au.dk/"^                  12
               unique_filename^"\" ><br>"))
    end;                                                              14
```

Figure 6.19: Create a plot using Gnuplot.

downloaded from a web server when a user requests an HTML document referring to the Java applet. When the Java applet is downloaded it is automatically started within the browser using a Java interpreter on the client machine. Using Java applets the simulator on the web server and the Java applet on the client machine can communicate during a simulation. By using Java applets it is possible to obtain interactive simulations via the web browser. To be able to use Java applets there are some extra requirements for the simulation tool related to communication between the Java applet and the simulator. It requires TCP/IP communication between the applet in the browser and the simulator residing on a server.

Finally, future work may also include developing auxiliary functions for generating templates of code for batch scripts and HTML forms. In particular template code for retrieving data from input fields in forms would be easy to generate automatically. The CPN expert could annotate the relevant system parameters in the CPN model and then the coloured Petri net tool could automatically generate a HTML form including fields for the annotated variables. Furthermore, it could also create functions for parsing the fields of the form and for assigning the system parameters in the CPN model to the values entered in the form by the application user. Experiments will show whether creating such template code will be useful in practice. One thing that will indeed be gained from generating both the HTML form and the functions for parsing the form is consistency between the HTML form and the CGI script, i.e. it will be possible to avoid some errors due to inconsistency between the names of the fields in the HTML form and the names referred to by functions for retrieving data from a form.

In conclusion, this paper has described a technique for making simulation of CPN models usable for people without knowledge of the technical details of CPN models. Thus, the paper has opened for using CPN simulators behind services on the Internet.

6.7 Acknowledgements

The ideas presented in this paper have been developed during a project conducted in cooperation with Søren Christensen, University of Aarhus, Lee W. Wagenhals, Insub Shin and Daesik Kim from George Mason University, Fairfax, VA, USA. We want to thank these people for valuable discussions during the development of the CGI approach. In particular, we want to thank Lee W. Wagenhals for his involvement in writing early versions of this paper. Finally, we also thank the CPN Group at University of Aarhus and the anonymous reviewers for their careful and valuable comments on this paper.

Chapter 7

Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic

The paper presented in this chapter has been published in the proceedings of the Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01).

[55] B. Lindstrøm and S. Haider. Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB–544*, pages 35–54. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/-CPnets/workshop01/.

Except for minor typographical changes the content of this chapter is equal to the original paper in [55].

Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic

Bo Lindstrøm* Sajjad Haider†

Abstract

This paper discusses coloured Petri net models of so-called timed influence nets with logic. Previous work has developed a method to translate a timed influence net into a coloured Petri net. The work in this paper describes a new and more compact translation from timed influence nets with logic into coloured Petri nets. The translation has the property that the net structure of the coloured Petri net will be the same for all translated timed influence nets – only the initial marking changes depending on the actual timed influence net. This more compact translation avoids the generation of simulation code for each timed influence net. The paper also presents some validation results to establish that the coloured Petri net models from the two methods are equivalent, i.e. that the new compact coloured Petri net model gives the same simulation results as the old less compact model does.

Key words: Coloured Petri nets, Timed influence nets with logic, Equivalence, Folding, Information assurance.

7.1 Introduction

When making decisions it may be very important to be able to maximise or minimise the probability of certain events to appear. Being able to solve this problem is relevant in several different areas. For command and control systems in the military it is, e.g. important to take the best possible actions to maximise the probability that an enemy will surrender. For a system administrator it is important to be able to minimise the down-time for some service.

This paper will use an example from the area of information assurance [92]. Information assurance aims to minimise the probability that an intruder breaks into a system and accesses secret information. Even though a lot of effort may be put into making systems secure, a system will not be more secure than the weakest point of the system. Therefore, it is very important to be able to analyse a system to find the probability that an intruder can access the secret information, and to find the weak points in a system.

*Department of Computer Science, University of Aarhus, Denmark.
E-mail: blind@daimi.au.dk.
†System Architectures Laboratory, George Mason University, Fairfax, VA, USA.
E-mail: shaider1@gmu.edu.

The method described in this paper makes it easier for an analyst to determine the probability that an intruder can access the secret information. It requires a model of the environment to be created by the analyst. The model is specified as a non-cyclic graph with probabilities to determine how likely an event is to appear. The model will be specified using a slightly modified version of influence nets [90, 21] which is called timed influence nets with logic (TINL). *TINL* is a variant of Bayesian nets [43] which makes it simple to specify probabilities. The logic of a TINL has nothing to do with temporal logic, instead the word logic refers to different kinds of nodes in a TINL. We will give more details on the definition of TINLs in Sect. 7.2.

Based on the specification of the TINL, coloured Petri nets (CP-nets or CPNs) [44, 45, 46, 48] are used to estimate the probability that a certain event occurs, and the time of the appearance. Most often the analyst is only interested in the final probabilities, and is not at all interested in the details of the CPN model. However, in some situations the details of how a simulation has evolved may be of interest, e.g. the order of occurrences of binding elements may contain useful information.

Previous research in this field [108] has developed a method to automatically convert a TINL into a CPN model using the tool Design/CPN [27]. Given a TINL specification file, the method automatically creates a CPN model which has a net structure very similar to the TINL, i.e. for each node in the TINL a specific part of the CPN model can be identified to model that node. Therefore, the fact that the structure of the CPN model looks like the TINL means that it is easy for an analyst to recognise nodes from the TINL in the CPN model. This close relationship between the CPN model and the TINL has been important when trying to convince analysts who are used to working with TINLs that CPN models are able to solve the problem and give the correct results.

In a later project the method has been extended to give access to creating and simulating models via a web browser. Given a TINL-specification file which is uploaded to a web server, the method automatically generates the CPN model and the corresponding simulator code using Design/CPN. Based on the simulator code, a CGI script [35] is automatically generated using the method described in [54]. Using an automatically generated web page which contains TINL-specific information, the user is able to set initial conditions for the simulator of the CPN model, and to start a batch of simulations using the CGI script. After having performed a number of simulations, results including graphics are displayed in the web browser. Figure 7.1 shows one of the graphs that are included on a web page to display the results to the analyst using domain-specific graphical user interfaces. The graphs show how the probabilities of certain nodes in the TINL evolve during time. Using this web-based graphical user interface, the analyst will never see the CPN model. Actually, the analyst does not even need to know that a CPN model is used to produce the results.

The previous method performs its job as expected. However, the method has some drawbacks. The fact that a completely new CPN model (with all the places and transitions) has to be generated for every influence net, means that it takes rather long time to apply the method. Even though the CPN model is

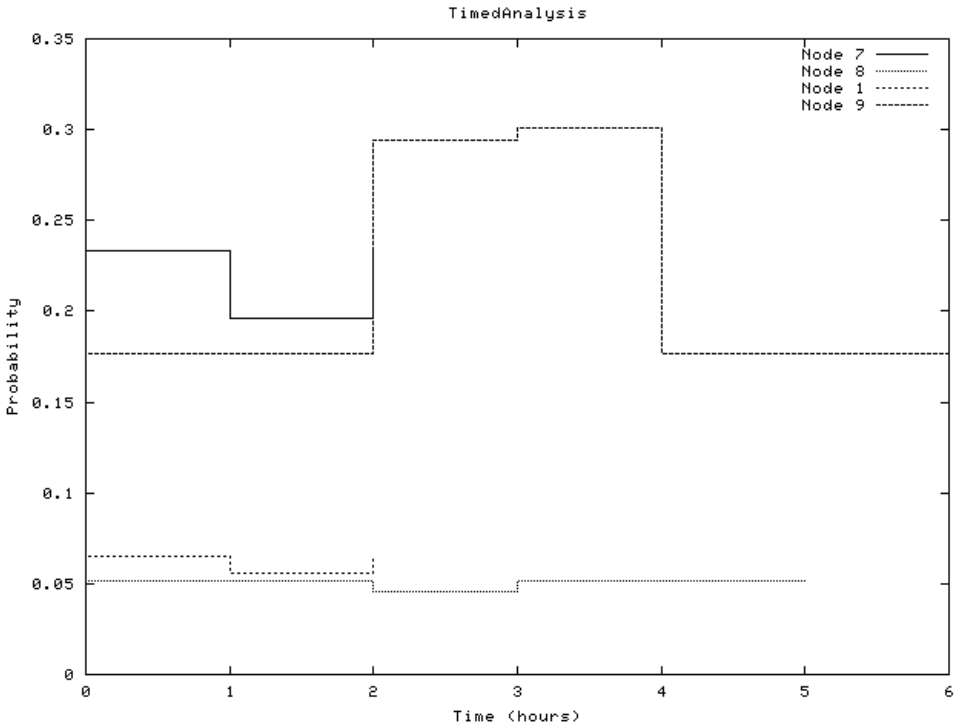


Figure 7.1: Graph from web page showing graphical analysis results of the TINL in Fig. 7.3.

generated automatically, it takes time both to generate the net structure and declarations, and then afterwards to generate the executable simulator code for that specific CPN model.

In this paper we describe how to avoid to generate simulator code for every instance of a TINL. In that way we eliminate the time used for each TINL to both create the specific CPN model and to generate the corresponding simulator code. This is done by creating a general CPN model (in the following referred to as the *folded CPN model*) which can simulate any TINL for any set of parameter values, i.e. the whole class of TINLs. The general CPN model models the actual computation or execution of a TINL, but it is not fixed to a specific TINL. The information of a specific TINL is maintained using colours, and not net structure as in the former project. That means that the folded CPN model is fixed to a specific influence net by using a specific initial marking which contains information about the net-structure of the specific TINL. In other words, the structure of the specific TINL is encoded in colours.

The old method which generates a CPN model with net structure similar to the structure of the TINL (in the following referred to as the *unfolded CPN model*), is used in combination with the folded CPN model. The folded CPN model is only used for automatic simulations, i.e. simulations controlled via a web browser, while the unfolded CPN model is used when a person with knowledge of TINLs has an interest in looking at the actual CPN model.

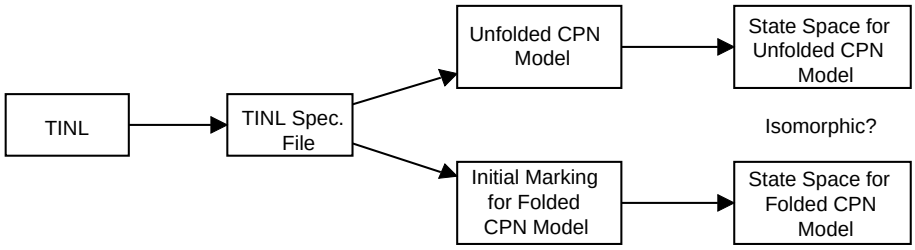


Figure 7.2: Overview of the applied method.

The fact that the behaviour of the two models is expected to be the same requires some validation. Figure 7.2 illustrates the method applied in this paper. From a TINL a specification file is generated to contain all static information about a TINL. From that file, the old method developed in the former project is used to generate the unfolded CPN model, and then the corresponding state space is generated. The TINL specification file is also used to initialise the marking of the folded CPN model to reflect the current TINL. The corresponding state space is also generated for the folded CPN model after the initialisation. Finally, the the equivalence of the behaviour of the two models has been validated by making a number of tests. This has been done by comparing the state spaces from the two CPN models.

The paper is structured as follows. Section 7.2 presents TINLs informally, and introduces an example which will be used for illustration purposes in the rest of this paper. Section 7.3 briefly describes how an unfolded CPN model will look like when it is generated using the old method. Section 7.4 describes the folded CPN model, and how it has been constructed from the unfolded CPN model. Section 7.5 discusses how we have validated that the behaviour of the two models are equivalent. Finally, in Sect. 7.6 we conclude and give directions for future work.

7.2 Timed Influence Nets with Logic

In this section we informally introduce timed influence nets with logic (TINL) by presenting an example in the field of information assurance. The example will be used as a running example in the following sections.

TINLs can be used to estimate the probability of a certain event when that event depends on other events which contains an element of uncertainty. Like e.g. Petri nets, TINLs have both a graphical and a mathematical representation. They are specified as directed acyclic graphs where the nodes represent events as random variables (a numerical quantity defined in terms of the outcome of a random experiment, p. 110 in [37]), while the arcs represent dependencies between the events. In addition timing information and probabilities are specified for the nodes.

Consider Fig. 7.3 which contains an example of a TINL. The purpose of the TINL is to estimate the probability that an intruder can unlock a door. To

unlock the door two options exist. It is possible to unlock the door using both a password and an access card. It is also possible to unlock the door using a physical key and the same password which is used with the access card. It is of interest to an analyst to know which of the two options is most likely to be broken by an intruder, and how long time it will take to do it.

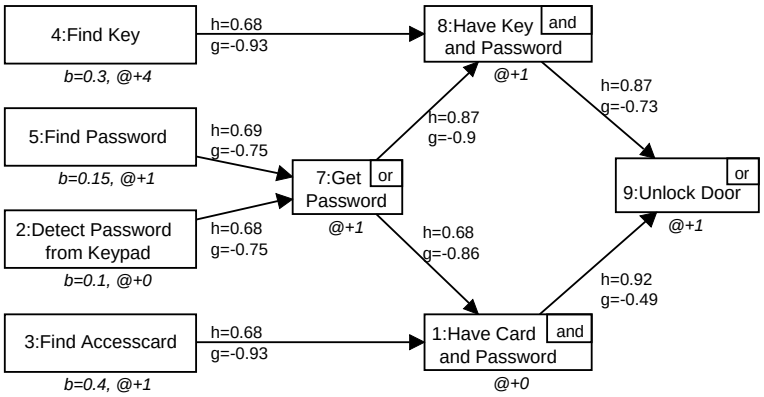


Figure 7.3: An example of an influence net for information assurance.

An intruder may be able to find either a physical key or an access card somewhere. The probability that the intruder finds a key is estimated to 30% (which is indicated by a so-called baseline probability *b=0.3* next to the node Find Key with number 4) while it is 40% for the access card (node number 3). The $h = 0.68$ on the arc to the node Have Key and Password indicates that there is 68% probability that the node Have Key and Password will be true if the node Find Key is true. Likewise $g = -0.93$ indicates that there is 93% probability that the node Have Key and Password will be false if the node Find Key is false. The time to find a key is estimated to 4 time-units, which is indicated by the expression `@+4` below the node Find Key.

The intruder is assumed to be able to obtain the password in two different ways. Either the intruder can find the password on a piece of paper Find Password with an estimated probability of 15%, or a special device can be used to detect the password from the keypad where the users types in the password Detect Password from Keypad with an estimated probability of 10%.

The node Get Password indicate the event that the intruder has obtained the password in some way, and therefore now knows the password. Notice that the node is a so-called *or-gate* which is reflected by the *or* in the upper right corner of the node. This indicates that we are only interested in the probability of the predecessor-event which has the highest probability, i.e. the post probability of the node is only to be based on the probability of the node with the maximal probability of all the predecessor nodes. Assume that n_7 denotes the random variable of the node Get Password and the predecessor nodes are denoted by p_2 and p_5. Then the probability of the or-gates is computed using the following formula:

$$P(n_7) = Max[P(n_7|p_2)P(p_2) + P(n_7|\neg p_2)P(\neg p_2), P(n_7|p_5)P(p_5) + P(n_7|\neg p_5)P(\neg p_5)]$$

Probabilities denoted by, e.g. $P(n_7|p_2)$ are so-called conditional probabilities. A conditional probability denotes the probability that a random variable (n_7) will be true when another random variable (p_2) is assumed to be true. Conditional probabilities are computed using the g and h values included in the TINL. We will not go into more details with how the probabilities are computed, but will refer to [103] for further details.

The two nodes Have Key and Password and Have Card and password indicate the events of having obtained two of the necessary items to unlock the door. These nodes are so-called *and-gates* which means that we require both predecessor events to be true. Therefore, the probability of the node will depend on both of the predecessor nodes, and not only one of them as it is the case for or-gate nodes. Consider node Have Key and Password and assume that n_8 denotes the random variable of the node and that the predecessor nodes are denoted by p_4 and p_7. The probability of and-gates is computed using the following formula (where e.g. $P(n_8|p_4, p_7)$ denotes the probability that n_8 is true when the two conditional random variables p_4 and p_7 are true):

$$\begin{aligned} P(n_8) = \quad & P(n_8|p_4, p_7)P(p_4)P(p_7)+ \\ & P(n_8|p_4, \neg p_7)P(p_4)P(\neg p_7)+ \\ & P(n_8|\neg p_4, p_7)P(\neg p_4)P(p_7)+ \\ & P(n_8|\neg p_4, \neg p_7)P(\neg p_4)P(\neg p_7) \end{aligned}$$

Again, the conditional probabilities can be computed based on the g and h values on the arcs from the predecessor nodes. Notice that the formula for computing the probability of an and-gate is more complex than the formula for or-gates due to the fact that it requires conditional probabilities for more than one predecessor, e.g. the conditional probability stated by $P(n_8|p_4, p_7)$.

Finally, the node Unlock Door indicate the event that we are actually interested in, i.e. to estimate the probability that an intruder is able to unlock the door using any of the legal possibilities. This node is also specified as an or-gate because we do not care which of the predecessor events are true. It is sufficient that one of them are true to be able to unlock the door.

When having specified probabilities for an influence net, an algorithm using the two formulas stated above can be used to propagate the probabilities forward through the net. The propagation algorithm starts by updating the initial nodes (nodes without predecessors). Then it updates the successor nodes using the newly computed values, and continues to update nodes until no more successor nodes exists. Therefore, for each initial node the probability is propagated through the TINL to the terminal nodes.

Figure 7.1 from Sect. 7.1 actually shows a graph showing the evolution during time of the probabilities of the intermediate and terminal nodes of the TINL presented in this section. We see that the probability for unlocking the door (Node 9) will be about 17.6 after time 4.

To complete this informal presentation of TINLs, we briefly relate TINLs to two closly related classes of nets which are also used to estimate probabilities.

TINLs are a variant of influence nets which again are a variant of Bayesian nets [43]. The difference is that for influence nets the predecessor nodes of a node are assumed to be independent from each other, while in Bayesian nets the predecessor nodes are not assumed to be independent. The independence of predecessor nodes in influence nets simplifies both the specification of probabilities in an influence net, and the actual computation of the probabilities. TINLs extend influence nets with the so-called or-gate, and with a time-concept.

7.3 Old Approach: Unfolded CPN Model

In this section we will briefly summarise how a CPN model of a TINL looks like when constructed using the method presented in [108]. However, the method has been slightly modified and extended to handle the or-gate discussed in Sect. 7.2 as well. Focus will be on the general pattern of a CPN model created from a TINL. We will not go into details with how to automatically generate the CPN model. Details on the automatic generation of a CPN model can be found in the paper [108].

Revisit the TINL in Fig. 7.3 in Sect. 7.2. Notice the three different types of nodes: initial (nodes without predecessors), intermediate (nodes with both predecessors and successors), and terminal nodes (nodes without successors).

Figure 7.4 presents the net-structure of an automatically generated CPN model for the TINL in Fig. 7.3. Each of the three types of nodes in the TINL is converted into one of three standardised subnets. Each rounded box models exactly one node in the TINL. Notice for later use in Sect. 7.4 the similar net-structure of the subnets, and that the subnets are connected only via places with colour-set Store.

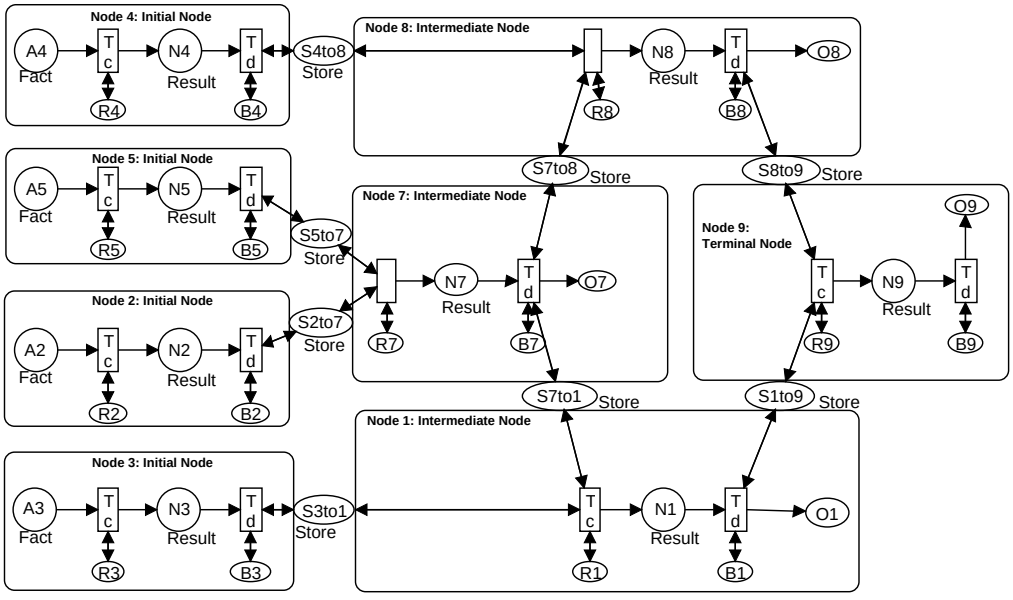


Figure 7.4: Net-structure of CPN model for the TINL in Fig. 7.3.

The number of predecessor and successor nodes of a given node may be different. Therefore, the number of Store places may be different for different subnets. Notice in Fig. 7.4 that the intermediate node number 8 has one successor node (S8to9) while intermediate node number 7 has two successor nodes (S7to8 and S7to1).

The model has two types of transitions and five types of places. The transitions named Tc are used to calculate new probabilities while the transitions named Td are used to distribute the newly calculated probabilities to the successor nodes. Figure 7.5 shows the details of the intermediate node number 7.

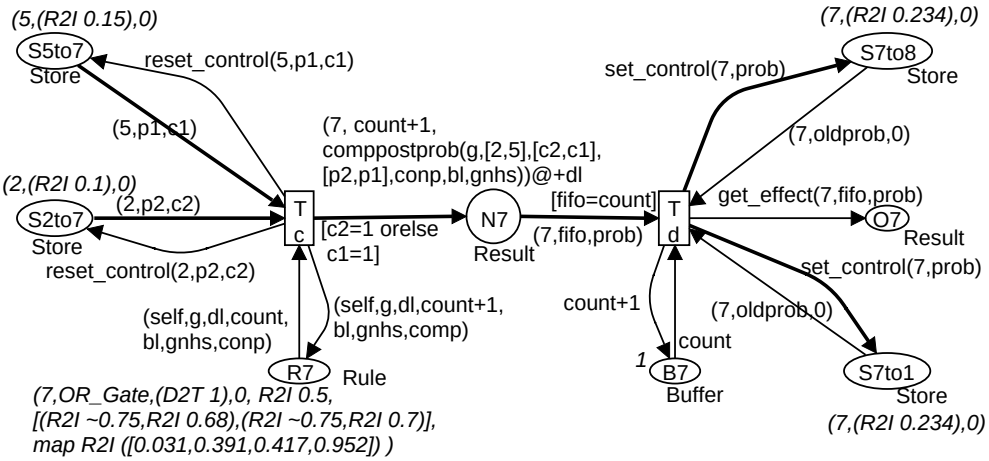


Figure 7.5: Details of intermediate node 7 in Fig. 7.4.

Places with the timed color-set Fact contain pairs of node id and the corresponding probability. These places are used to model the initial probabilities of a node (also called initial beliefs).

Places with color-set Store has a similar purpose as the Fact places. Like colour-set Fact, the colour-set Store contains a node id and a probability. However, in addition it contains a boolean control value which is used to indicate whether the probability of that store has been recalculated by the predecessor node and not yet been processed by the successor node. This control value is used to propagate newly computed values forward.

The timed color-set Result is also a triple. It contains a node id, a sequence number, and the probability of the node. The purpose is to capture the resulting probabilities so that the analyst can inspect the values of the tokens when a simulation ends.

Color-set Rule is a 7-tuple. First of all, it contains the node id. The second value is the gate value indicating whether the node is an and-gate or an or-gate. The third value is the time-delay of the node which indicate how much time it takes to perform the action represented by the node. The fourth value is a counter used to give tokens a sequence number to be used for "first in, first out" handling of tokens. The fifth value is the baseline value of the node. The sixth value is a list of the g and h values described in Sect. 7.2. Finally, the last value is a list of conditional probabilities. The values in the list gives all

the conditional probabilities stating how likely the node is to be true when any subset of the predecessors are true.

Finally, color-set SeqNo is a counter or sequence number which is used to ensure a first in, first out protocol when distributing tokens to the successor nodes.

As mentioned above, the transition Tc calculates the probability of the node given the probabilities of the predecessors. The function `comppostprob` computes the value based on probabilities from the predecessor stores and the values at the Rule place. However, for the intermediate and terminal nodes, the guard of the transition prevents the transitions from being enabled unless the probability of at least one predecessor node has been recalculated. That is done using the control value of the tokens on the predecessor Store places. When the tokens are put back to the Store places, the control value is reset to indicate that the probability has been propagated forward. This is the essential control mechanism of the forward propagation algorithm used for TINLs.

The transition Td is used to distribute computed probabilities to the successor nodes, and to the result place. The transition gets the next probability from the place N in fifo order, i.e. the token which has first arrived to the place N is first removed. In addition the transition removes the old probability from the successor-Store places, and replace these tokens with the new probability, and sets the control variable of these tokens to indicate that a new probability has been calculated.

The most important part of this section is to understand the general pattern used for constructing subnets for the individual nodes, and that any CPN model of a TINL can be created by combining these subnets into a complete CPN model. The detailed behaviour of the model is of secondary importance. In the next section we will describe the general or folded model which does not have so much net structure, but is instead able to model any TINL.

7.4 New Approach: Folded CPN Model

The method described in [108] generates a new CPN model for every different TINL as discussed in Sect. 7.3. In this section we describe a CPN model which can simulate any TINL when initialised with a proper marking. The CPN model (folded CPN model) is created as a folding of the CPN model (unfolded CPN model) presented in Sect. 7.3. This has several advantages: it is easy to validate that the folded CPN model has behaviour equivalent to the behaviour of the unfolded CPN model; when one knows the details of one of the models it is easy to learn the details of the other model; and it is easy to modify and maintain the models.

7.4.1 Hierarchy Page

The folded CPN model is created as a hierarchical CPN model with the hierarchy page depicted in Fig. 7.6. The CPN model has two prime pages. The page InitModel is used to load and set the initial markings of the CPN model, and will be discussed in the end of this section. The page Top is the top-page of the

model which gives an overview of the CPN model. The three pages Initial Node, Intermediate Node, and Terminal Node models the three different types of TINL nodes.

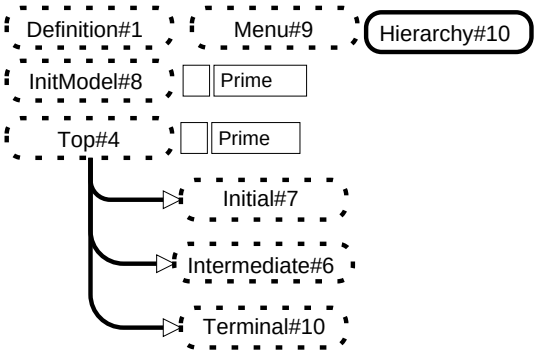


Figure 7.6: Hierarchy page for the folded CPN model.

7.4.2 Top Page

After being initialised, the transition Driver on the top page of the CPN model in Fig. 7.7 starts to activate the simulation of the TINL nodes. First at least one initial node must be activated, next all three kinds of nodes may be activated. The left part (left of the substitution transition Initial Node) and the output place O is similar to the same page in the unfolded CPN model except that in the unfolded CPN model, one output places existed for each node with output. In other words, we have folded several output places from the unfolded CPN model into a single output place O in the folded CPN model. The colour-set Result includes the id of the node. That implies that it is possible to distinguish tokens belonging to different nodes even though they are stored on a single output place.

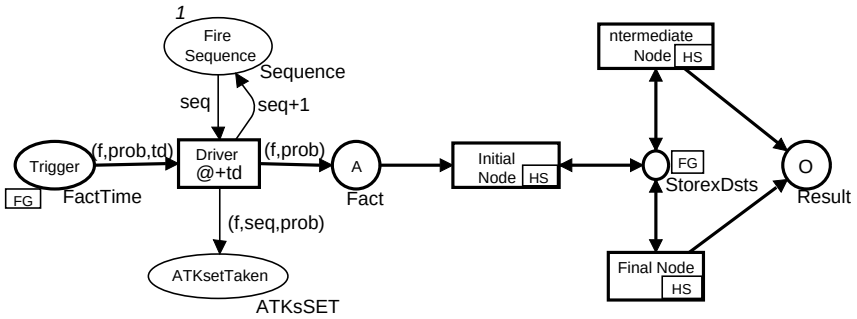


Figure 7.7: Top page of folded CPN model.

7.4.3 Folding Initial, Intermediate, and Terminal Nodes

Revisit Fig. 7.4 in Sect. 7.3. Notice that the structure of all initial nodes (nodes 2-5) is the same. Each initial node consist of four places and two transitions. The structure of the intermediate nodes (nodes 1, 7, and 8) is also the same like it is for terminal nodes (node 9). Only the number of Store places representing connections between nodes may be different.

Let us consider intermediate nodes in more detail. As mentioned above, all intermediate nodes have the same net structure. That means that we can fold all intermediate nodes into a single general folded node. Figure 7.8 shows the CPN model of the folded intermediate node which is a generalisation of nodes like Fig. 7.5.

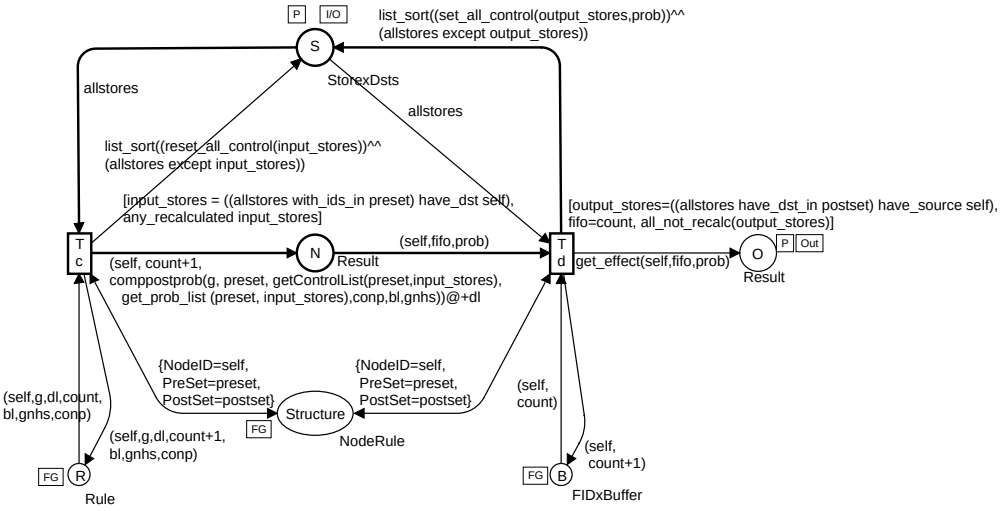


Figure 7.8: Folded intermediate page.

Folding Places

The folded intermediate node has been obtained by folding all similar places and transitions. In this section we focus on folding places, while we consider folding transitions in Sect. 7.4.3.

Revisit Fig. 7.4 and focus on the places of the intermediate nodes 1, 7, and 8. Places with the same colour-set are folded into one place. E.g. the rule places (R_i, where i is the node id) are folded into one place (R in Fig. 7.8) representing rules for all intermediate nodes.

The actual folding has been done using the well-known method of adding a unique identity to each token to indicate which place in the unfolded CPN model the token belongs to. The colour-set Rule already contains the node id which makes the tokens from different nodes distinguishable or unique. In a similar manner the place N has been obtained by folding all places N_i, and the place O by folding places O_i. The place B in the folded CPN model is also a folding of all B_i places from the unfolded model. However, this colour-set has

been modified to include the node id to make it possible to distinguish tokens from different nodes.

The colour-set StorexDsts of the place S in the folded model is the colour-set which has been changed the most. It is indeed the folding of all places S$i$to$j$, where i and j are respectively the ids of the predecessor node and the successor node. The colour-set Store in the unfolded model already contains the id of the predecessor node. However, as discussed above, a node may have several successors (node 7 has both nodes 1 and 8 as successor nodes in Fig. 7.4). Therefore, the id of the successor node (or destination) has been added to the colour-set StorexDsts to be able to distinguish tokens belonging to different successor nodes. For reasons related to the effectiveness of calculating enablings during simulations in Design/CPN, the colour-set is changed to a list of tokens. In summary, the place S contain a list of tokens from the places S$i$to$j$ from the unfolded CPN model which are made unique by adding the id of the successor node to each token.

We have added one new place which does not exist in the unfolded CPN model. It is the place Structure. It contains information about how intermediate nodes are connected to other nodes, i.e. structural information about how intermediate nodes are connected in the TINL. The colour-set NodeRule is a record with three values. The value `NodeID` is the id of the node. The values `PreSet` and `PostSet` contains lists of the ids of the predecessor and successor nodes of the current node. As an example, a token representing all the arcs to/from the Store-node 7 in Fig. 7.5 is: `1‘{NodeID=7,PreSet=[2,5],PostSet=[1,8]}`. The nodes 2 and 5 are predecessors of node 7 while nodes 1 and 8 are successors. Using this place it is possible to encode all arcs between nodes into colours, and by initialising this place with tokens containing the appropriate information we can model different dependencies between nodes in a TINL.

Folding Transitions

Let us now consider the transitions `Tc` and `Td` in Fig. 7.8 for intermediate nodes. Like for the places, these two transitions are foldings of all intermediate `Tc` and `Td` transitions in the unfolded CPN model in Fig. 7.4. When a transition occurs in the folded model it reflects the behaviour of the occurrence of exactly one transition in the unfolded model.

Let us consider how transition `Tc` can be enabled. The simulator starts by taking a random token from the places R and Structure which has the same node id (`self`). The variable `allstores` on the arc from the place S to transition `Tc` binds to the single list on the place S.

Now the guard of transition `Tc` is evaluated. First the variable `input_stores` is assigned the list of values returned by evaluating the expression `((allstores with_ids_in preset) have_dst self)`. This expression runs through the list of all the stores and finds the stores with ids in the `preset` variable from the Structure place which have destination (or successor) `self`. Now the variable `input_stores` contains a list of the stores of the predecessor nodes of the currently considered node. Next, the expression `any_recalculated stores` tests if any of the `input_stores` have the control value set to indicate that

they have been recalculated by predecessor nodes. This function corresponds to the guard of `Tc` in the unfolded CPN model. If the guard (the function `any_recalculated`) evaluates to true, the transition can occur.

When the transition `Tc` occurs a token is put on the place `N` to indicate that the node is in the process of being updated. The function `comppostprob` is exactly the same function as used in the unfolded model (see the corresponding arc expression in Fig. 7.5). By using the same function in the folded model, we avoid introducing errors by writing new functions for computing the probabilities. The functions `getControlList` and `get_prob_list` returns the control and probability lists from the `input_stores`. The control values of the `input_stores` are reset using the function `reset_all_control` on the arc from `Tc` to `S`, and appended to the unchanged stores (`allstores except input_stores`). The resulting list is sorted using the function `list_sort` to make a canonical representation of the list. The reason is related to generating the state-space for the model which is used for validation in Sect. 7.5. If we do not sort the list, the order of interleavings of transitions in the CPN model will have impact on e.g. the multi-set bounds of the place `S`.

Consider transition `Td` which distributes the newly computed probabilities to the `S` place. First the transition match the token on place `N`, the counter-token on place `B`, and the token on the place `Structure` with the corresponding node id. It also binds the variable `allstores` to the list on place `S`. A guard somewhat similar to the guard on transition `Tc` calculates the `output_stores` corresponding to the node which is currently considered. The expression `((allstores have_dst_in postset) have_source self)` returns the list of all the stores with node id in the `postset`-list of the node – but only those which have source or node id `self`. The expression `fifo=count` in the guard ensures fifo-handling of the tokens on place `N`. This is similar to the guard in the unfolded model. Finally, the function `all_not_recalc` tests if all of the `output_stores` have control value equal to 0. This corresponds to the `0` in Fig. 7.5 on the arc from $\mathsf{S_7to_8}$ to `Td`.

When transition `Td` occurs it updates the probability and the control field of the corresponding `output_stores`, and returns the list to the place `S` added to the remaining stores.

The folding of the initial and terminal nodes is conducted in a way similar to the one described above for intermediate nodes, and is therefore not included here.

7.4.4 Initialisation of the CPN Model

The fact that the folded CPN model is to simulate any TINL means that the model has to be initialised with appropriate data to reflect the structure of a specific TINL. This data is loaded into the model from a TINL-specification file. This file is generated using the CAT-tool[1] and contains all data needed to create the initial marking. The file is exactly the same file as the one used to create the unfolded model as described in details in [108].

[1]CAT is the Effects Based Campaign Planning and Assessment Tool under development at the US Air Force Research Laboratories (AFRL/IF).

Figure 7.9 depicts the page where tokens are computed and distributed to the places of the CPN model. The transition Distribute Tokens is the only transition being enabled initially. When it occurs, several things happens. First, the code segment of the transition loads the TINL-specification files for initialising the CPN model. Then multi-sets of tokens are generated using functions like `createBufferTokens` in Fig 7.10. Given a list of node ids from the TINL-specification file, the recursive function generates for each node id, a token `1‘(nid, 1)`. The rest of the markings for the remaining places are generated using similar more or less complex functions.

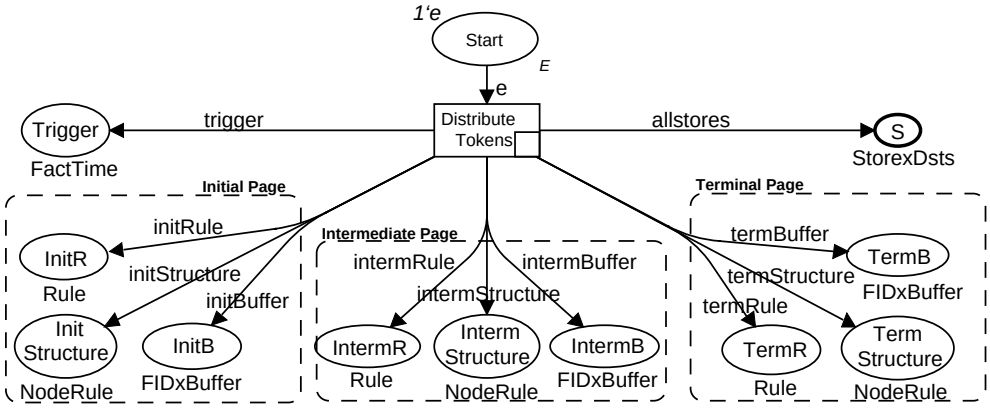


Figure 7.9: Loading and distributing initial tokens.

```
fun createBufferTokens ([]): FIDxBuffer ms =  empty
  | createBufferTokens (nid::rest) =
    1‘(nid,1) ++ (createBufferTokens rest);
```

Figure 7.10: Create tokens with nid and a counter with initial value 1.

The distribution of the tokens is done by means of fusion places. For example, the places IntermR, IntermB, IntermStructure, and S are fused with the corresponding places (R, B, Structure, and S) on the intermediate page in Fig. 7.8. In this way tokens are loaded and computed on a single page of the CPN model, but are distributed to the places on several pages of the CPN model.

7.5 Validation

In this section we describe how we have validated that the folded CPN model and the unfolded CPN models gives the same simulation results. The folded CPN model is constructed from the unfolded CPN model in a way which makes us expect that the state spaces from the two CPN models will be equivalent.

The fact that the folded CPN model can model any TINL by being initialised with different initial markings means that we have a whole class of different CPN

models. That means that when we have an initial marking of the folded CPN model, then we have one unfolded CPN model which has the same behaviour, and vice versa.

We could have conducted a mathemathical proof to prove that the two CPN models have equivalent behaviour. This would ensure that the equivalence holds for any TINL. However, we have chosen to use state space analysis to check that for the given TINLs, the equivalence is likely to hold. We will show that given a specific initial marking, the behaviour of the folded CPN model is equivalent to the behaviour of the unfolded CPN model. We will focus on sizes of the state spaces, boundedness properties, and paths in the state spaces. We have used the state space tool of Design/CPN for the analysis to be presented in this paper.

7.5.1 Model Similarities and Statistical Information of the State Space

Before we go into details with how the equivalence has been validated, we will briefly mention a few important issues related to how the folded CPN model has been obtained from the unfolded CPN models.

As mentioned in Sect. 7.4, when we have a transition in the folded CPN model, then this transition is the folding of a number of transitions in a unfolded CPN model. In other words, when we have a binding of a transition in the unfolded model, exactly one binding of a transition exists in the folded CPN model. The only exception from the correspondence between transitions in the folded and the unfolded CPN models is the transition Distribute Tokens depicted in Fig. 7.9 in Sect. 7.4. This transition has no counterpart in the unfolded CPN model. However, it is the first transition to occur in the initial marking of the folded CPN model, and it will only be enabled once. That means that the impact on the state space is the addition of exactly one node and one arc to the state space for the folded CPN model. However, to avoid this extra node we first let the transition Distribute Tokens occur, and then start to generate the state space. That implies that we have the exact same number of nodes and arcs in the two state spaces. When we refer to the state space of the folded CPN model in the following, we will refer to the state space without the first node.

The fact that the folded CPN model has been constructed in the way mentioned above, makes it easier to check the equivalence of the folded and unfolded CPN models. The reason is that the number of nodes and arcs in the state space for each of the two models are equal, i.e. when there is one node in one of the state spaces, then there will be a similar node in the other state space, and similar for arcs. Therefore, ignoring this first node and arc from the state space makes the state spaces equivalent for a given initial marking of the two CPN models.

As an example, consider the TINL depicted in Fig. 7.3 in Sect. 7.2. We have generated the state space for both the unfolded and the folded CPN models. By generating the state space report for this example we obtain the statistical information in Table 7.1. The state space for the folded CPN model contains

exactly the same number of nodes as the unfolded CPN model, and likewise for arcs. This gives us strong evidence to believe that the structure of the state spaces are identical.

Unfolded CPN model		Folded CPN model	
Nodes:	13930	Nodes:	13930
Arcs:	29595	Arcs:	29595
Secs:	301	Secs:	642
Status:	Full		Full

Table 7.1: Statistics for the state space for the TINL in Fig. 7.3.

We have also generated the strongly connected component graphs (SCC-graphs) for both CPN models, and they have exactly the same number of nodes and arcs as the corresponding state spaces. This was expected because TINLs are acyclic directed graphs, and thus the state space of the corresponding CPN model is to be an acyclic graph.

The pairwise equivalent nodes in the two state spaces are expected to have the same number of input and output arcs. In other words, from a given marking in either of the two CPN models, the same number of binding elements can be concurrently enabled. We have checked that this is the case by counting the nodes with the same number of input and output arcs {(0 input arcs, 0 output arcs), (0 input arcs, 1 output arc), ..., (n input arcs, n output arcs)}. As expected, it turned out that we got the same numbers from both CPN models. This check gives us further reason to believe that the statical structure of the TINL represented in the folded CPN model is correct.

7.5.2 Boundedness Properties

In this section we will focus on the results or markings produced by the CPN models – rather than the structure of the state spaces. We will show that for representative examples the two models give the same results. We will focus on the output places of the CPN models because the marking of these places are the ones showing the actual results of interest to the person simulating the model. In addition, the markings of these places are highly correlated with the correctness of the CPN models because the markings are based on the forward propagation of probabilities from the initial parameters.

Integer Bounds

First we consider integer bounds. Integer bounds give information about the maximal and minimal number of tokens which may be located on the individual places within the reachable markings. We can use that to check if the number of tokens in the folded and unfolded CPN models have the same bounds.

Table 7.2 shows the upper and lower integer bounds for output places (O_i) of the unfolded CPN model for our example TINL. We are able to use integer bounds only because the markings of these places are monotonically increasing, i.e. tokens are added but no tokens are removed from these places. From the

Place	Upper Bound	Lower Bound
PN1'O1	3	0
PN1'O7	2	0
PN1'O8	3	0
PN1'O9	6	0

Table 7.2: Unfolded CPN model: integer bounds.

Place	Upper Bound	Lower Bound
Top'O	14	0

Table 7.3: Folded CPN model: integer bounds.

table we see that in total there can be up to 14 (2+3+3+6) tokens on the output places during a simulation (in our case it will be at the end of a simulation), while the lower bounds are 0 for all output places which means that they are empty initially.

Table 7.3 shows the bounds for the folded CPN model. This model has only one output place which is the folding of the four places of the unfolded CPN model. We notice that the upper bound is 14 like the sum of the upper bounds for the unfolded CPN model. Also for this model the lower bounds is 0. Now we know that the bounds of the output places in the two models are the same which means that we are more confident that the two models gives the same number of outputs when given the same input parameters.

Multi-set Bounds

We will now focus on the actual values of the tokens in the two models. The multi-set bounds give us information about the values which the tokens may carry. By definition, the upper multi-set bound of a place is the smallest multi-set which is larger than all reachable markings of the place. We consider the multi-set bounds for the output places. From the upper multi-set bounds we will be able to see which probability values may be calculated during executions of the two models.

Compare Tables 7.4 and 7.5 containing the multi-set bounds for the output places in the two models for our example TINL. Notice that e.g. the place PN1'O7 in the unfolded CPN model contains two tokens with node id 7, sequence numbers 1 and 2, and probabilities 196500 and 233750. From the upper multi-set bounds for the place Top'O in the folded CPN model we see that these tokens will also be present in this model. By comparing the rest of the upper multi-set bounds we see that they are equal as well.

We have also included multi-set bounds for the places Structure in the folded CPN model (see Fig. 7.8 for details). From these bounds we see that the tokens describing the structure of the TINL are indeed as expected. This can be observed by comparing preset and postset of the individual nodes with the structure of the corresponding unfolded CPN model in Fig. 7.4.

Place	Best Upper Multi-set Bounds
PN1'O1	1‘(1,1,(ii (”56131”)))++ 1‘(1,1,(ii (”65095”)))++ 1‘(1,2,(ii (”56131”)))++ 1‘(1,2,(ii (”65035”)))++ 1‘(1,3,(ii (”65035”)))
PN1'O7	1‘(7,1,(ii (”196500”)))++ 1‘(7,2,(ii (”233750”)))
PN1'O8	1‘(8,1,(ii (”45673”))) ++ 1‘(8,2,(ii (”51935”)))++ 1‘(8,3,(ii (”51935”)))
PN1'O9	1‘(9,1,(ii (”294572”)))++ 1‘(9,1,(ii (”300891”)))++ 1‘(9,2,(ii (”171766”)))++ 1‘(9,2,(ii (”294572”)))++ 1‘(9,2,(ii (”300849”)))++ 1‘(9,3,(ii (”171766”)))++ 1‘(9,3,(ii (”176807”)))++ 1‘(9,3,(ii (”300849”)))++ 1‘(9,4,(ii (”171766”)))++ 1‘(9,4,(ii (”176807”)))++ 1‘(9,4,(ii (”300849”)))++ 1‘(9,5,(ii (”176807”)))++ 1‘(9,6,(ii (”176807”)))

Table 7.4: Unfolded CPN model: best upper multi-set bounds.

7.5.3 Equivalent Paths in State Spaces

In addition to checking the equivalence of the behaviour of the two models only by means of the above mentioned techniques, we have tried to compare paths in the state spaces of the two models. We have checked that, whenever one of the models can make a step (let a transition occur) then the other model must also be able to make a step. This was done by exploring paths through the state space. We considered the arcs on the path in the state space of the unfolded CPN model. For every binding of a transition on the path in this CPN model, we checked if the corresponding folded transition in the folded CPN model was enabled. This was indeed the case for the paths that we followed. In addition, we checked that for every node on the path, the same number of successor nodes existed in both CPN models. From this comparison we get even more confident in the equivalence of the two CPN models.

However, we plan to do this more consequently than what we have done so far. We will define a mapping M_{unfold} from states in the folded CPN model into states in the unfolded CPN model. This mapping should be based on the equivalence between markings in the folded and unfolded CPN models. A similar mapping BE_{unfold} should be defined for mapping binding elements in the folded CPN model into binding elements in the unfolded CPN model. Then the two mappings M_{unfold} and BE_{unfold} should be applied to the state space of the folded CPN model to show that resulting state space is identical to the state space of the unfolded CPN model (modulo the extra initialisation marking and arc in the folded CPN model).

7.6 Conclusion and Future Work

When using CP-nets for creating models one have to be careful to choose the right level of folding. This paper shows that it is sometimes useful to have

Place	Best Upper Multi-set Bounds
Top'O	1‘(1,1,(ii (”56131”)))++ 1‘(1,1,(ii (”65095”))) ++ 1‘(1,2,(ii (”56131”)))++ 1‘(1,2,(ii (”65035”))) ++ 1‘(1,3,(ii (”65035”)))++ 1‘(7,1,(ii (”196500”)))++ 1‘(7,2,(ii (”233750”)))++ 1‘(8,1,(ii (”45673”))) ++ 1‘(8,2,(ii (”51935”)))++ 1‘(8,3,(ii (”51935”))) ++ 1‘(9,1,(ii (”294572”)))++ 1‘(9,1,(ii (”300891”)))++ 1‘(9,2,(ii (”171766”)))++ 1‘(9,2,(ii (”294572”)))++ 1‘(9,2,(ii (”300849”)))++ 1‘(9,3,(ii (”171766”)))++ 1‘(9,3,(ii (”176807”)))++ 1‘(9,3,(ii (”300849”)))++ 1‘(9,4,(ii (”171766”)))++ 1‘(9,4,(ii (”176807”)))++ 1‘(9,4,(ii (”300849”)))++ 1‘(9,5,(ii (”176807”)))++ 1‘(9,6,(ii (”176807”)))
Initial'Structure	1‘{NodeID = 2,PreSet = [],PostSet = [7]}++ 1‘{NodeID = 3,PreSet = [],PostSet = [1]}++ 1‘{NodeID = 4,PreSet = [],PostSet = [8]}++ 1‘{NodeID = 5,PreSet = [],PostSet = [7]}
Intermediate'Structure	1‘{NodeID = 1,PreSet = [3,7],PostSet = [9]}++ 1‘{NodeID = 7,PreSet = [2,5],PostSet = [8,1]}++ 1‘{NodeID = 8,PreSet = [7,4],PostSet = [9]}
Terminal'Structure	1‘{NodeID = 9,PreSet = [1,8],PostSet = []}

Table 7.5: Folded CPN model: best upper multi-set bounds.

co-existing models with different degrees of folding.

One CPN model with a relatively extensive net structure to be used for understanding the model and for reference to the modelled system. This may require that the level of folding is kept so low that a new CPN model has to be created for each instance of the problem. However, if the model can be generated automatically as for the models presented in this paper, it may not be a problem.

The second model should focus on modelling the entire class of problems. This model is typically a folded version of the other one, which implies that the one and only model can be used for any instance of the problem. That means that the general CPN model will typically have relatively little net structure compared to the specific models.

A method for validating the equivalence between the folded and unfolded models has successfully been applied to several different TINLs with different time delays. Therefore, we believe in that the behaviour of the models are equivalent. However, we have not proved or verified that they are equivalent.

Future work will first of all focus on verification of the equivalence of the behaviour of the models. In particular we want to get more confident that the structure of the state spaces from the different models are equivalent. We plan to establish a formal proof to prove the equivalence of the behaviour of the

models.

We will also experiment with alternative graphical interfaces for the simulators of the CPN models. The current web-based graphical interface is only useful for non-interactive simulations. By being able to interact with the simulator during simulations it will be possible to access the current state of the TINL represented by the CPN model during a simulation. This may be useful if the simulator is integrated with the CAT-tool (using TCP/IP communication) which is used to create the TINL. That would make it possible to display the results in the CAT-tool during simulations.

Acknowledgements. This research was conducted at the C3I Center of George Mason University (GMU), VA, USA, with partial support provided by the U.S. Air Force Office for Scientific Research under Grant No. F49620-98-1-0179. The work in this paper proposing or-gates in influence nets for information assurance is based on work done by Insub Shin, GMU. Special thanks goes to Alexander H. Lewis and Lee W. Wagenhals, GMU, for valuable discussions and support during the project. Finally, we also want to thank Kurt Jensen and the anonymous referees for constructive critique and usefull suggestions on improving the paper.

Chapter 8

Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets

The paper presented in this chapter has been published in the proceedings of the Workshop on Formal Methods Applied to Defence Systems which was held as a satellite event at the Petri Nets 2002 conference in Adelaide, Australia.

[56] B. Lindstrøm and L. Wagenhals. Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets. In C. Lakos, R. Esser, L. Kristensen, and J. Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 115–124. Australian Computer Society Inc., June 2002.

Except for minor typographical changes the content of this chapter is equal to the original paper in [56].

Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets

Bo Lindstrøm* Lee W.Wagenhals†

Abstract

We present an environment for web-based simulation of influence nets to be used for operational planning. Previous work in this field has shown how influence nets, which are used for probabilistic modelling, can be extended with time and then translated into a coloured Petri net to do temporal evaluation of plans. Simulating the coloured Petri net model in a web environment makes it easy for a subject matter expert to use the simulator for planning without knowing the underlying coloured Petri net formalism and tools. This paper discusses the use of influence nets for operational planning, a simulator for influence nets implemented using coloured Petri nets, and the architecture of the complete web-site which can be used for operational planning.

Keywords: Operational planning, Courses of action, Effects based operations, Influence nets, Coloured Petri nets, Web-based simulation.

8.1 Introduction

Planning an operation to achieve objectives can be a very complex task. A plan may depend on interplay between several different complex events, where only some of the events are controllable. In many cases, plans are developed to try to compel an adversary to take actions or make decisions that the adversary is not pre-disposed to make.

The operational planning method presented in this paper has been motivated by a concept called effects based operations (EBO). EBO is the notion

*Department of Computer Science, University of Aarhus, Denmark.
Email: blind@daimi.au.dk.
†System Architectures Laboratory, George Mason University, VA, USA.
Email: lwagenha@gmu.edu.

of selecting actions based on their collective contribution to desired and undesired effects. The set of selected actions is called a course of action (COA). To support EBO, at least two problems must be addressed. The first is to relate effects to actionable events. In this problem, we need to define the set of desired and undesired effects on the adversary. From these effects we work backwards, from effects to causes, and identify actions which we believe to have impact on the effects. Finally, we arrive at the actions that we have at our disposal for achieving the effects. In the second problem, called the COA problem, we must select from the set of all possible actions those subsets that will yield, with high probability, the effects we wish to achieve, and with low probability the undesirable effects. Then, taking into consideration constraints associated with specific actions or combinations of actions, the selected actions must be sequenced and time phased. This timing of the actions is important to be able to know in which order and when the controllable actions should be done to obtain the best possible conditions for a successful operation. The result is a set of alternative COAs. These COAs are then evaluated against requirements to determine the COA that provides the best likelihood of causing the desired effects to occur and the undesired effects to not happen.

Probabilistic models like, e.g. Bayesian inference nets [43] and influence nets [90], can be used to model a situation to determine which actions to take to optimise the outcome of an operation. However, these models do not include the temporal aspect of the planning.

Previous research on influence nets is presented in, e.g. [108] and has demonstrated how timing information can be added to influence nets by translating the influence net and a timing profile to a coloured Petri net (CP-net or CPN). This CP-net can then be simulated both to estimate the probability of a successful operation, and to evaluate a plan for timing the operation. The translation from the influence net and the timing profile to the CPN model is done completely automatically, i.e. the places, transitions, and declarations of the CPN model are generated without user interaction. In this paper we will refer to a CP-net generated using this method as a *fixed CPN model*.

The influence net formalism has been extended slightly to include so-called logical gates. An influence net with logical gates together with timing information is in the following called a *timed influence net with logic* (TINL).

Recent work, presented in [55], has developed a *generic CPN model* that is able to simulate any TINL. In other words, a *TINL CPN simulator* has been implemented. The advantage of the generic CPN model is that time is saved during the translation, because a new CPN model does not need to be generated each time a new TINL needs to be examined as it had to when generating fixed CPN models. Instead, the generic CPN model within the TINL CPN simulator is initialised with appropriate tokens to reflect the concrete TINL. One issue of concern was that the simulation time for the generic TINL CPN simulator would be much longer due to the complex colour sets and potential large number of tokens in places required by the generic CPN model. Experiments presented in this paper has shown that using a more effective CPN simulator has overcome this problem.

It is essential that tool support is available to and usable by (1) the team of

intelligence analysts and other subject matter experts (SMEs) responsible for analysing a situation in terms of actions and effects and (2) operational planners who perform resource allocation and scheduling to cause actions to occur. A modelling tool that can be used by these teams called CAESAR II/EB [106] has been built to focus on the belief and reason aspects of an adversary so that potential actions can be related to effects. The tool incorporates influence nets as the probabilistic modelling technique and the TINL CPN simulator to support the temporal aspects of COA evaluation. These two formalisms enable the modeller to create the structure of actions, effects, beliefs, decisions, and the influencing relationships between them.

A *two stage operational concept* that uses the two modelling techniques to perform COA analysis has evolved through tests in realistic scenarios [106]. Figure 8.1 illustrates the two stages in applying the CAESAR II/EB tool. In the first stage, intelligence analysts and SMEs develop an influence net to specify the probability of effects given sets of actions. Once the influence net has been created, it is exported to an influence net specification file so that operational planners can perform temporal evaluation in stage two. The goal of stage two is to determine and recommend the timing of the set of actions that give the best set of acceptable probabilities for all effects. In stage two, a delay file for the influence net is created by the operational planner to specify a full TINL. Then temporal analysis can be conducted to analyse different COAs by simulating the TINL using the TINL CPN simulator. The simulation results provide, for a given timed sequence of actions, the probability of effects over time, in the form of a probability profile [105]. Probability profiles indicate how long it will take for a specific COA to achieve the desired effects, reveal time windows of risk when the probability of effects is unacceptable, and provide time windows for indicators of success or failure. Changing the timing of selected actions can significantly change the probability profiles.

Thus, a tool like CAESARII/EB can be used to support an overall planning process in which a situation analysis team creates a model with the tool that relates potential actions to overall effects that can be used by a team of operational planners that perform the resource allocation and scheduling function to evaluate plans. The operational planners execute the model using the tool to evaluate proposed plans for acceptability and may adjust the timing of actions within resource constraints until acceptable outcomes in terms of the timed phased probability of effects is achieved.

One problem of translating an influence net to a CP-net is that the analysis is conducted using another formalism than the one which is used for specifying the influence net. In practice, neither the intelligence analyst nor the SME necessarily knows the CPN formalism or the CPN tools. Therefore, the translation puts an extra, unnecessary skill-requirement onto the SME. As a consequence, an alternative domain-specific graphical user interface (GUI) for the TINL CPN simulator has been developed, and is presented in this paper. The GUI is created using simple so-called HTML documents and CGI scripts [35], and is based on the tool Design/CPN [27] and on the method presented in [54]. Simulating the CPN model using a domain-specific GUI makes it easy for the SME to use the simulator for planning without knowing the underlying CPN formalism and

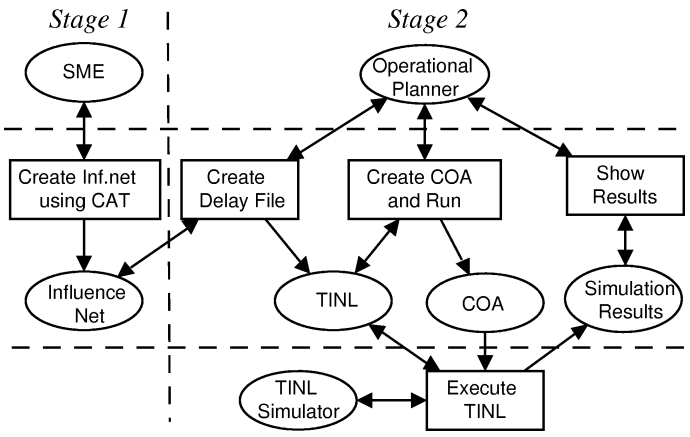


Figure 8.1: Two-stage operational concept.

tools. The main objective of this paper is to illustrate how to create and use a domain-specific GUI in a web-based environment for simulating TINLs to be used for operational planning.

The paper is structured as follows. Section 8.2 describes how influence nets and the TINL CPN simulator can be applied in practice for operational planning. Section 8.3 discusses the generic CPN model. Section 8.4 describes the web-site used to control the TINL CPN simulator. Section 8.5 compares the speed of executing alternative CPN models that have been proposed for simulating TINLs. Finally, Sect. 8.6 concludes and gives directions for future work.

8.2 Practical Use of the Operational Concept

In this section we describe the two stages of the operational concept. First we describe stage 1 by introducing influence nets and discussing how an influence net can be created to support effects based operations. Next, in stage 2 the execution of an influence net using the TINL CPN simulator from the web environment is described.

8.2.1 Stage 1: Creating Influence Nets

Influence nets, a variant of Bayesian nets, have been used since 1994 [52, 90] to depict the causal relationships between actions and events. They are acyclic directed graphs. A small example of an influence net with six nodes is shown in Fig. 8.2, and will be used for illustration in this paper. The modelled situation is that Fed wants to set the conditions so that Borg will not use the weapon of mass destruction called a death star (DS) against Fed in a war. The leftmost nodes in an influence net represent actionable events which are events that can be controlled. The rightmost nodes represent the decisions or effects that are either desired or undesired – seen from the point of view of the SME. In the middle are the nodes relating actions to effects.

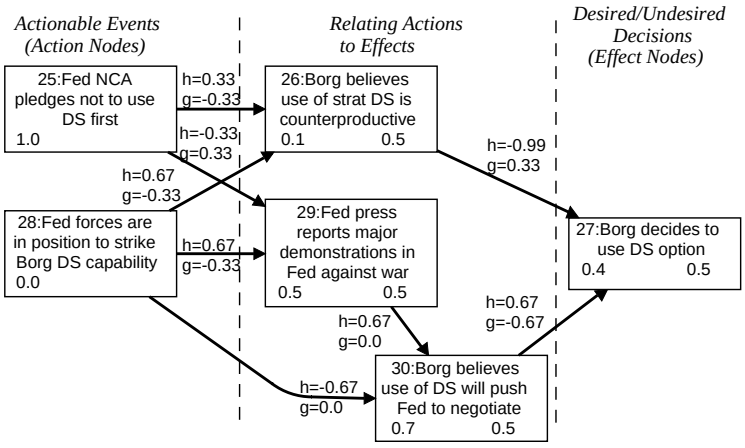


Figure 8.2: Example of an influence net.

The nodes in an influence net represent statements or beliefs with which a probability value can be associated. For example, the upper-middle node 26 represents the statement that "Borg believes use of strat DS is counterproductive". Each node with parents has a parameter called the *baseline probability* which indicates the probability that the event can be true independently of all other modelled events. The baseline probability is positioned in the lower right corner of the nodes with parents in an influence net (0.5 for node 26).

A directed arc represents a directed binary relationship between two nodes. Two numbers, called influences, characterise the relationship and are denoted by h and g. The first one, h, represents the strength of the influence that a parent node has on a child node, if the parent node was to be true. The second one, g, reflects the strength of the influence if the parent node was not true. Both h and g can take values in the closed interval [-1, 1] which means that the binary relations can be either promoting (+) or inhibiting (–). For example, the SME believes that node 25 (upper left) has impact on node 26. In the actual application, the analyst generally uses a set of qualitative statements that map to a set of values in the interval [-1, 1]. The typical statements and their corresponding values are as follows. Significantly less likely: -0.99, moderately less likely: -0.67, slightly less likely: -0.33, no effect: 0, slightly more likely: 0.33, moderately more likely: 0.67, significantly more likely: 0.99. For example, to set an h value, the analyst will select an answer from the question: if the parent were to be true, the impact on the child would be to make it e.g. "slightly more likely" (select the best statement from the list). In the example, the analysts estimate is that when node 25 is true, then the node 26 will be slightly more likely to be true (thus $h = 0.33$), and when node 25 is false, then node 26 will be slightly less likely to be true (g = -0.33).

After an analyst has created an influence net and assigned the values of the g, h, and baseline probability parameters throughout the net, the values are translated into conditional probabilities for each node with parents using an algorithm called CAST (for Causal Strength) [12]. The influence net can then be used to propagate probabilities from the leftmost nodes with no parents

(action nodes) to the rightmost nodes with no children (effect nodes). In this paper we have used the CAT-tool[1] to create influence nets. In addition, the CAT-tool has been used to automatically convert the influences (h's and g's) between parent and child nodes to the conditional probabilities via the CAST algorithm.

Once the influence net has been completed, it can be used to evaluate the impact of actions on the effects (decisions) of interest. This can be accomplished by sensitivity analysis or by executing the influence net. The sensitivity analysis consists of finding those actionable events that have most impact on the effects of interest. This analysis makes it possible to use only those actionable events which have the most impact on the effect nodes. To execute the influence net, the analyst sets the probabilities of a set of actionable (leftmost) events to either zero or one, depending on whether the action is planned or not, and evaluates the influence net. Then the tool propagates these probabilities from left to right until all effects are accounted for in the rightmost nodes representing main effects. An analyst can experiment with the influence net by changing the probabilities of one or more of the actionable events and seeing what the effect is on the key decision nodes.

Once the specification of the influence net has been completed, and the actionable events have been selected then the influence net is exported from the CAT tool to a so-called influence net specification file. This file is a textual representation of the influence net with the static information which is necessary to simulate an influence net. The file is then transferred to a web server where it is used by a script to specify the initial markings of the generic CP-net within the TINL CPN simulator.

8.2.2 Stage 2: Temporal Evaluation using CP-Nets

The next activity involves the operational planners who assess the availability of resources to carry out the tasks that will result in the occurrence of the actionable events. The resulting plan will indicate when each actionable event will occur. Selecting the set of actions that will lead to achieving the overall desired effects while not causing the undesired ones is not the only important task. Determining the timing of those actions is critical to achieving the desired outcomes. It is not possible to evaluate the impact of the timing in the CAT-tool because influence net does not contain temporal information. However, because influence nets assume the independence of causal influences, it is possible to associate time with either the nodes or the arcs of the influence net. The time delays represent the amount of time it takes for knowledge about a change in the status of any node to be propagated by some real world phenomenon to the node that is affected by that change. The update in the marginal probability (the current probability) of a node occurs immediately after the time delay. It is these time delays, along with the timing of the actions, that causes the generation of probability profiles which are graphs depicting the probability of each node as a function of time.

[1] CAT is the Effects Based Campaign Planning and Assessment Tool developed by the US Air Force Research Laboratories (AFRL/IF) and George Mason University.

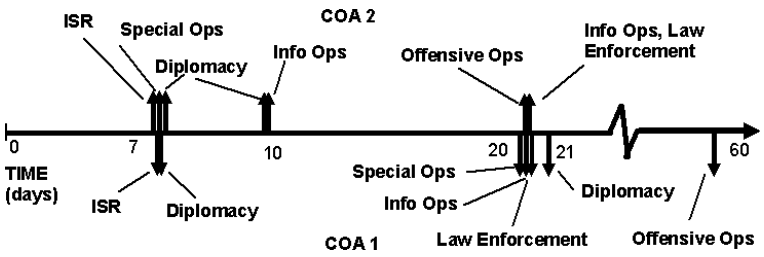


Figure 8.3: Two COAs with the same actions but different time-phases.

In this paper we will consider only two types of temporal information. The first one is the timing of the actionable events, i.e. the specification of when the controllable actions should be carried out. The second type of delay is the estimated time it takes before the consequence of each event represented in the influence net can be seen. This second timing information is expected to be constant for several COAs. An example of these two types of timing information is that the event represented by node 25 in Fig. 8.2 may be initiated after two hours and then the effect of the event will be seen after three hours.

The input scenario is described in terms of the actions chosen from the set of actionable events, in the selection of the COA and the time at which these actions occur. The actions are modelled as events, which means that they occur instantaneously. To give an idea of different scenarios, an example with two COA scenarios is shown in Fig. 8.3. The actions and their timing of *COA1* are indicated below the time line while the same set of actions with different timing that comprise *COA2* is shown above the time line.

Let us now turn to how the CAESAR II/EB tool is used for temporal evaluation. An analyst uses a web browser to specify the temporal information and to simulate the influence net under a variety of initial conditions. This is done by filling out a set of HTML forms that are automatically generated using a so-called CGI script [35]. A CGI script is a program that runs on a web server, and it can be started, for example, by submitting an HTML form from a web browser.

First, the temporal information must be specified by the analyst. The influence net specification file from the CAT tool is used by a CGI script to automatically generate an HTML document, where the operational planner can input the time delays of the individual nodes. Figure 8.4 contains such an HTML document for the influence net in Fig. 8.2. The HTML forms contains an entry for each node in the influence net with the text from the corresponding node in the influence net. When the form is submitted, the information is saved, and can later be accessed using the delay name specified in the form.

Once the time delays have been specified, the operational planner is ready to analyse different COAs using the TINL CPN simulator. This is done by first requesting the generation of a new HTML document where the input scenario can be specified. Figure 8.5 contains an example of such a document. The document contains two forms where input can be given. The first form is for specifying the timing for the actionable events that should be included in the

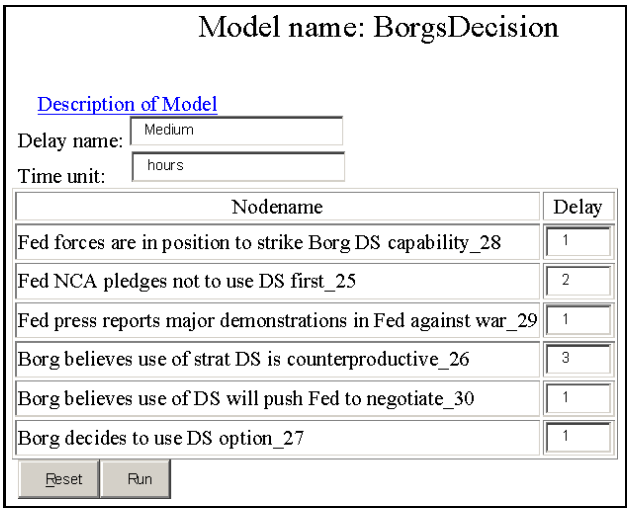


Figure 8.4: Web page to specify time delays.

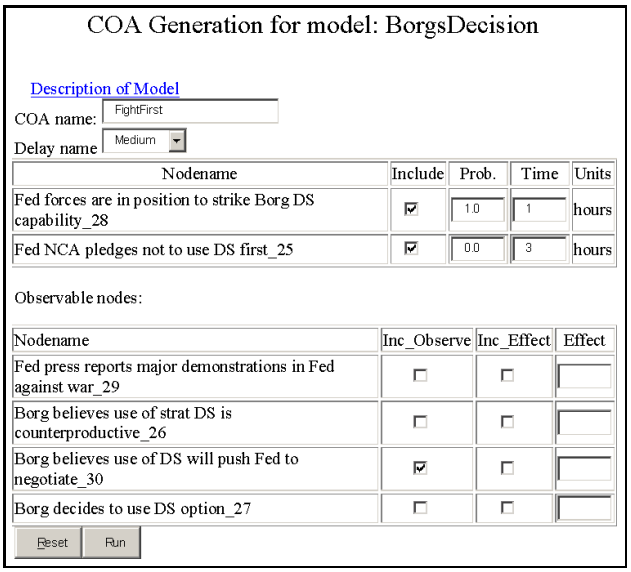


Figure 8.5: Web page to specify a COA and to select nodes for display of probability profile.

COA. The second form is used for two things. First, the operational planner must specify if there is strong evidence that some of the events are already true. For those events, the effect is set to 100% in the form. Secondly, it is specified for which of the nodes, the probability profile should be included in the graph of the probability profile.

When the HTML document is submitted, the web server sets the initial marking of the generic CPN model within the TINL CPN simulator and simulates the CPN model. During the simulation, the server automatically stores the values needed to plot the probability profiles in a file for later use in compar-

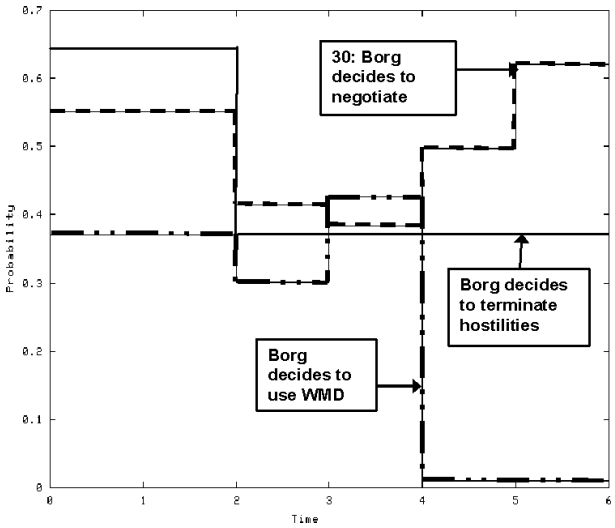


Figure 8.6: Plot of probability profile.

ing COAs. When the simulation ends, graphs are generated which contain the probability profiles that show the marginal probability for the selected nodes in the influence net as a function of time, and are then displayed in the web browser. The probability profiles indicate how long it will take for the effects of the actionable events to affect various nodes in the influence net and time windows when probabilities may have unacceptable values. By changing the timing of the actions in the COA, the analyst may be able to eliminate these unacceptable windows. The analyst will most likely concentrate on the probability profiles of the key decision nodes, i.e. the nodes with no children.

An example of three probability profiles for a single COA is shown in Fig. 8.6. The graphs contains plots of an extended version of the influence net for the Borg-example. The annotation boxes in the plot have been added manually for clarity. For this COA, the probability profile for node 30 indicates that after 5 hours Borg wants to negotiate with a probability of about 62%. The likelihood that Borg will decide to use weapons of mass destruction (WMD) decreases and then increases before it finally reaches a very low value. This indicates that there is some risk associated with this COA. The analyst will attempt to find an alternative timing scheme that will reduce this risk caused by the rise in the likelihood of WMD use.

To compare multiple COAs the analyst can fill out another HTML form in the web browser to generate plots that show the probability profiles of nodes for different COAs. Figure 8.7 contains an example of such an HTML document for the Borg example. In the first form a single node is selected, and two COAs are selected in the second form. When the HTML document is submitted, the probability profiles for the selected node and COAs will be displayed based on the data that was collected when the simulations were initially made. The ultimate output of the analysis of multiple COAs is a recommendation, along with the supporting rationale, for a particular COA.

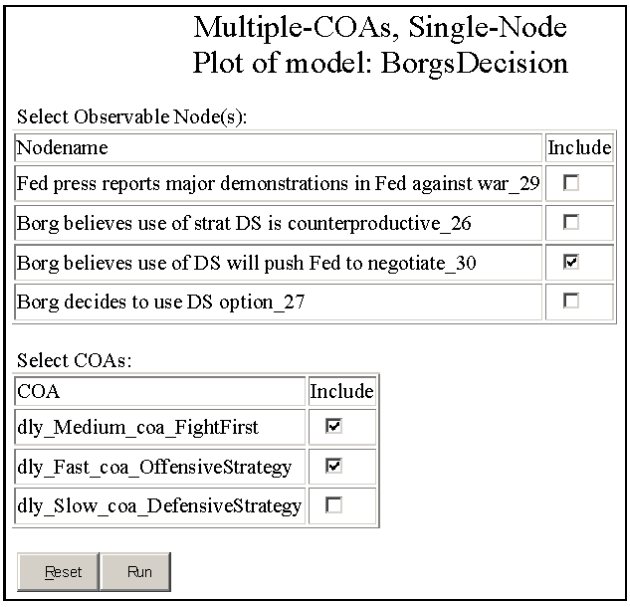


Figure 8.7: Web page to select nodes and multiple COAs for comparison of probability profiles.

In rapidly evolving situations, it is typical for analysts to continually modify the influence net model as new information about the situation becomes known. With the original CAESAR II/EB tool using fixed CPN models, each time a change was made in an influence net, a new CP-net had to be automatically created and compiled before any temporal analysis could be done. This can be time consuming in a situation where speed of analysis is important. The new technique of creating a CP-net that provides a generic representation of any influence net means that changes in the influence net do not require the generation of a CP-net. The implementation of this generic CP-net and more details on the web environment are discussed in the next sections.

8.3 CPN Simulator for TINLs

In this section we describe the generic CPN model [55] that can simulate any TINL when initialised with a proper marking. We remind that, in this context, a TINL consists of an influence net and timing information for the influence net. The purpose of this section is to illustrate how a simulator of TINLs can be implemented using CP-nets. Next, Sect. 8.4 will discuss technical issues on the architecture of the web-based environment for the TINL CPN simulator.

8.3.1 Motivation for the Generic CPN Model

In [108] a translation from TINLs to CP-nets has been developed. The translator takes a TINL specification file as input and automatically generates a fixed CPN model which can simulate that specific TINL. It is expected that the SME

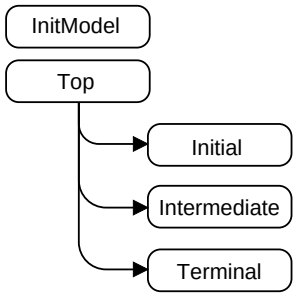


Figure 8.8: Overview of the generic CPN model.

wants to analyse several TINLs and COAs within a short time frame. Therefore, the time used to apply the method is of great importance. The most time consuming part of the method described in [108] is that for each TINL that needs to be investigated, a complete CPN model needs to be generated with places, transitions, declarations, etc., and afterwards the simulator code should also be generated. As we will see in Sect. 8.5, the turn-around time for generating the CPN model, the simulator code, and then running the simulation is relatively high for non-trivial TINLs. Even though a CPN model and the corresponding simulator code of the old translation method can be generated completely automatically, it is still a relatively time consuming job to apply the method with the tools currently available.

Based on these experiences, the generic CPN model has been developed. The advantage of this CPN model is that the simulator code is generated once, and only once. In other words, we not only avoid generating a new CPN model for each TINL, but we also eliminate the need for generating simulator code for each TINL. The generic CPN model which is used to simulate any TINL, is constructed such that its behaviour is equivalent to the fixed CPN models which are generated using the old translation method. The equivalence has been investigated in [55].

8.3.2 Overview of the Generic CPN Model

The generic CPN model is a hierarchical CPN model. An overview of the modules in the CPN model is given in Fig.8.8. The module InitModel is used to load the TINL specification file and the other parameters into the CPN model, and will be discussed in the end of Sect. 8.3.5. The Top module of the CPN model gives the most abstract view. The three modules Initial, Intermediate, and Terminal model three different types of TINL nodes. Initial nodes represent actionable events, i.e. nodes without predecessors, while the terminal nodes represent the decisions nodes, i.e. nodes without successors. The intermediate nodes represent the remaining nodes with both predecessor and successor nodes.

8.3.3 Top Module

The net structure in the CP-net models the flow in the probability propagation algorithm, while colours are used to model the nodes, arcs and probabilities

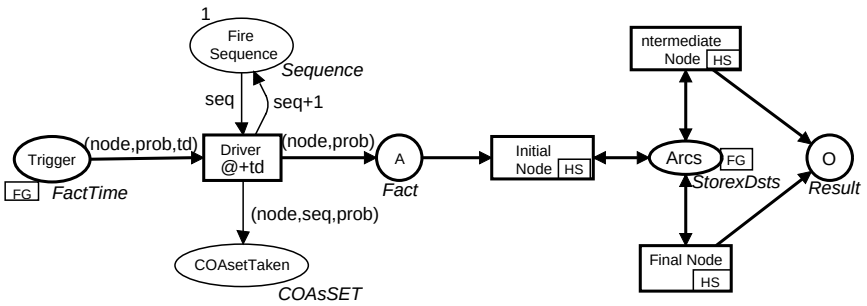


Figure 8.9: Top module of the generic CPN model.

of a specific influence net. Therefore, most of the colour sets include the node identity so that it is possible to distinguish tokens belonging to different nodes even though they are located at the same place. For example, in the Top module, shown in Fig. 8.9, the leftmost place Trigger contains triples of node identity, the corresponding probability and initial time delay. The initial delays and probabilities for each of the nodes representing actionable (or controllable) events have been given as input in the HTML forms in Figs. 8.4 and 8.5 via the web browser.

Let us now turn to how the CPN model simulates a specific influence net. The transition Driver in the Top module starts by removing tokens from the Trigger place and adds tokens to the place A with the initial time delays (`td`). That corresponds to activating the actionable events for each `node` at different points in time. Next, the probabilities can be propagated forward from the initial nodes via the intermediate nodes to the terminal nodes.

8.3.4 Intermediate Nodes

Let us consider how the nodes of a TINL are simulated by investigating intermediate nodes only. The initial and terminal nodes are modelled in a similar way and will, therefore, not be discussed here. All intermediate nodes are modelled using the same net structure. Figure 8.10 shows the CPN model for intermediate nodes. The model has two transitions and six places. The general idea of this module is that the transition Compute Prob models the computation of a new probability based on probabilities of the predecessor nodes, while the transition Distribute Probs models the distribution of the newly calculated probability to the successor nodes.

Places and Colour Sets

In this section we discuss how the static information of a TINL is modelled by describing some of the places and colour sets of the intermediate module. In the next section we will consider the modelling of the probability propagation of a TINL.

Consider the place Arcs which models the arcs in a TINL. This place always

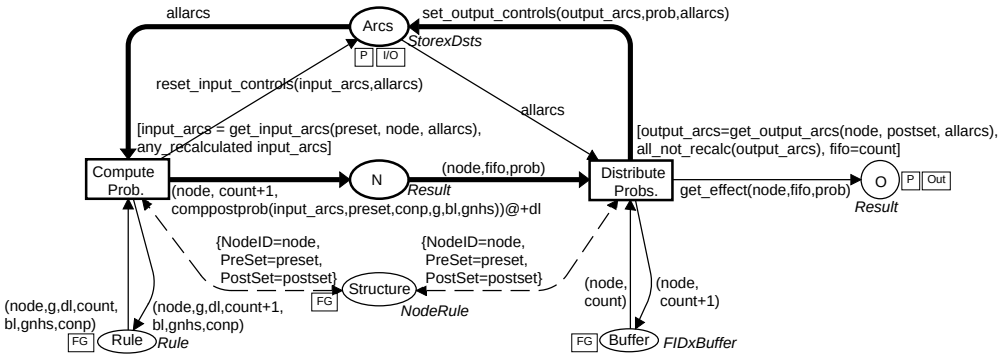


Figure 8.10: Intermediate module of the generic CPN model.

holds a single token which is a list containing an entry for each arc in the TINL. Each arc entry contains a source and a destination node id, a probability value, and a boolean control value. The control value indicates whether or not the probability of the predecessor node of the arc has been recalculated and is ready to be processed by the successor node. In other words, the control value is used to check if newly computed probabilities have been propagated forward or not. To model the arcs in Fig. 8.2 in Sect. 8.2 a list with eight entries is needed, i.e. one entry for each pair of connected nodes.

Apart from modelling the relationships between nodes, the static information of each node in the TINL must also be modelled to be able to compute the probability of each node. For that purpose, the place Rule has a token for each node in the TINL. To model a specific node, first of all, it contains the node identity of the node in the TINL so that it is possible to identify the token for a given node. The remaining values are the gate type (not discussed here), the time delay of the node from the delay file, a counter, the baseline probability, the h and g values, and a list of conditional probability weights stating how likely the node is to be true when any subset of the predecessors are true. Based on these values, it is possible to compute the post probability for a node when new probabilities arrive at the input arcs to the node.

Let us now turn to the place Structure. It contains static structural information about how each intermediate node is connected to other nodes in the TINL. The purpose is, for a given node to have an easy way to find the input and output arcs from the list of arcs on the place Arcs. For that purpose, the place holds a token, for each node, with a list of identities of the input (`PreSet`) and output (`PostSet`) nodes. As an example, a token representing arcs to node 26 in Fig. 8.2 in Sect. 8.2 from predecessor nodes 25 and 28, and the arc from node 26 to successor node 27 is: `1‘{NodeID=26,PreSet=[25,28],PostSet=[27]}`. Using such tokens it is possible to encode the arcs of each node into colours, and by initialising the place with tokens containing appropriate information we can model different connections between nodes in a TINL.

To model the fact that the events in a TINL takes a certain amount of time, the place N is used. This place holds newly computed probability values until the time elapses and then the probability can be propagated forward.

Transitions

Now the modelling of the dynamics of a TINL is discussed. When a node has propagated a new probability value forward to a successor node, the successor node must then recalculate its own probability, and then also propagate its new probability value forward to its own successor nodes, etc. This is modelled using the transitions Compute Prob. and Distribute Probs. in Fig. 8.10.

First consider when a node must compute its probability. It must be computed whenever one of its predecessor nodes has changed its probability. In terms of the CPN model, this means that the transition Compute Prob must occur when any of the input arcs to a given node has the control value set to one. To model this, the simulator first takes one token from each of the places Rule and Structure with the same node number (`node`). The variable `allarcs` on the arc from the place Arcs to transition Compute Prob binds to the single list on the place Arcs. Now the guard of transition Compute Prob is evaluated. First the variable `input_arcs` is assigned to the list of input arcs for the `node` by selecting those arcs from `allarcs` in the TINL that have endpoint in the in `node`. These arcs are exactly those having source in `preset` and destination in `node`. Next, the function `any_recalculated` tests if any of the `input_arcs` have the control value set to indicate that they have been recalculated by predecessor nodes.

When the transition Compute Prob occurs, a token is put on the place N with a time delay `dl` to indicate that the node is in the process of being updated. The probability of the node is calculated using the function `comppostprob` based on the probabilities of the input_arcs, and the static data on the place Rule. Finally, the control values of the `input_arcs` are reset using the function `reset_input_controls` on the arc from Compute Prob to Arcs to indicate that the probability has been propagated forward.

When the time delay has elapsed, the probability value must be forwarded to the successor nodes. In the CPN model this means that the `output_arcs` of the `node` must be updated. The transition Distribute Probs first matches a token on place N, a counter-token on place Buffer, and a token on the place Structure with the corresponding `node`. The guard calculates the `output_arcs` from list `allarcs`, based on the list `postset` for the given `node`. This is exactly those arcs which connect the `node` with its successor nodes. The function `all_not_recalc` ensures that all of the `output_arcs` have not set the control value. This is done to ensure that the probabilities have been propagated further on.

When transition Distribute Probs occurs it updates the probabilities, sets the control value of the corresponding `output_arcs`, and then returns the list to the place Arcs.

8.3.5 Initialisation of the CPN Model

The generic CPN model has to be initialised with appropriate markings to reflect the structure of a specific TINL. The data is loaded into the CPN model from the TINL-specification file, which is exported from the CAT-tool and contains all data needed to create the initial marking. The file is exactly the same file as the one used to create the fixed model as described in [108].

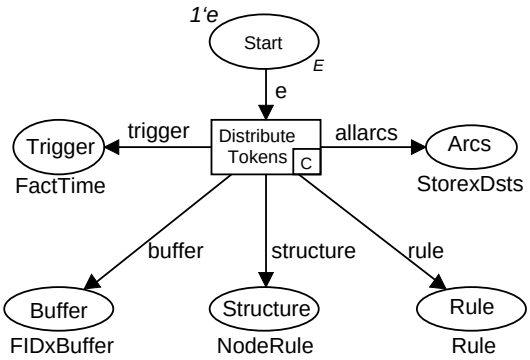


Figure 8.11: Loading and distributing initial tokens.

```
fun createBufferTokens ([]): FIDxBuffer ms =  empty
  | createBufferTokens (nid::rest) =
    1'(nid,1) ++ (createBufferTokens rest);
```

Figure 8.12: Create tokens with a node id and a counter with initial value 1.

Figure 8.11 depicts the module where tokens are created and distributed to the places of the CPN model. The transition Distribute Tokens is the only transition being enabled initially. When it occurs, several things happens. First, the code segment of the transition loads the TINL-specification files for initialising the CPN model. The filename is read from the HTML form used to start the simulation. Then multi-sets of tokens are generated using functions like `createBufferTokens` in Fig 8.12. Given a list of node ids from the TINL-specification file, the recursive function generates for each node id `nid`, a token `1'(nid, 1)`. The rest of the markings for the remaining places are generated using similar functions.

The distribution of the tokens is done by means of fusion places. The places Buffer, Structure, Rule, and Arcs are fused with the corresponding places on the intermediate module in Fig. 8.10, and the initial and terminal modules. The place Trigger is fused with the one in Fig. 8.9. In this way tokens are loaded and computed in a single module of the CPN model, but are distributed to the places of several modules of the CPN model.

8.4 Web-Based Simulation Environment

To get an impression of the complexity of the web-site for the TINL CPN simulator, this section first describes the relatively complex structure of the web-site and describes how results can be produced via the web environment. Secondly, the actions for controlling the TINL CPN simulator are discussed. Finally, we discuss why we found using HTML forms and CGI scripts to be a good choice for this project.

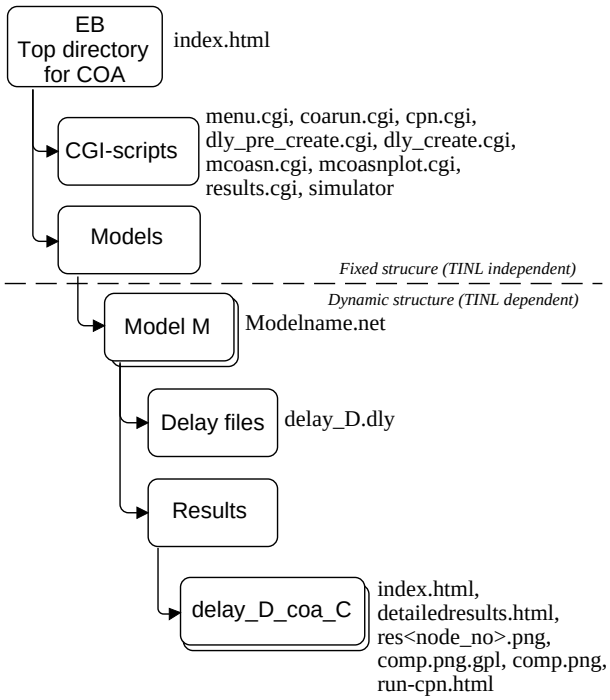


Figure 8.13: Directory structure for the web-site.

8.4.1 Structure of Web Environment

The directory structure on the web server is depicted in Fig. 8.13. The topmost directories are fixed and TINL independent. The remaining directories depend on the concrete TINLs, and are created and maintained automatically by CGI scripts while the user interacts with the GUI of the web browser.

The topmost directory is the `EB` directory where the HTML document `index.html` is located. This file is the entry to the web-site. In addition to this file, two directories (`CGI-scripts` and `Models`) are present.

The directory `Models` holds a directory (`Model M`) for each of the TINLs that is examined using the TINL CPN simulator. Therefore, when a new TINL is added to the web-site, a directory is created in the `Models` directory, and the TINL-specification file is put in this new directory. The directory `Delay_files` holds each of the delay profiles that are created from the web environment. Results of simulating this specific TINL are put in the `Results` directory. A new directory is created for each pair of delay files and COAs (`delay_D_coa_C`). The files in the `delay_D_coa_C` directories are all generated automatically from the TINL CPN simulator while simulating the given TINL. The purpose of the files is to make the simulation results persistent as HTML documents, graphics, and raw data. Later it is possible to query the persistent results and to display the results as HTML documents.

The directory `CGI-scripts` serves to hold all CGI scripts for the web-site and the TINL CPN `simulator` for the generic CPN model. The CGI scripts

are all simple Perl scripts [109]. We will not give all details of the CGI scripts here, but only the overall idea of what they are used for.

The CGI script `models.cgi` is used to create an HTML document which gives overview and access to each of the existing TINL models. When a specific TINL has been selected from the menu, the CGI scripts `dly_pre_create.cgi` can be used to create an HTML form where the operational planner can specify delay information of the nodes in the TINL. An example of this HTML form is shown in Fig. 8.4 in Sect. 8.2. When the form is submitted, the CGI script `dly_create.cgi` creates a delay file which can be used to specify delays of the nodes during a simulation.

The CGI script `coarun.cgi` is used to create HTML forms for specifying a COA. An example of such HTML forms is depicted in Fig. 8.5 in Sect. 8.2. In the HTML forms, probabilities and delays of actionable events (initial nodes) can be specified. When the form is submitted, the CGI script `cpn.cgi` executes the TINL CPN simulator by invoking the batch script contained in the TINL CPN simulator. The batch script will be discussed in Sect. 8.4.2. The results of the simulation are displayed as an HTML document which e.g. includes graphs like Fig. 8.6 in Sect. 8.2.

At any point in time, the simulation results which have been generated during a single COA of a previous simulation can be inspected. This is done using the CGI script `results.cgi` which gives access to the index files `index.html` for the individual COAs in the `delay_D_coa_C` directories.

After simulating several COAs and thereby generated simulation results for each of the COAs, it is useful to be able to compare these COAs with each other. The "multiple COAs, single node" approach is used to inspect simulation results from several COAs with respect to a single node. The CGI script `mcoasn.cgi` can generate an HTML form which is used to select a single node and some of the existing COAs. An example of output from this CGI script was given in Fig. 8.7 in Sect. 8.2. Afterwards, the CGI script `mcoasnplot.cgi` is executed to find the simulation results from the selected COAs, and then plot probability profiles of the node for these COAs. In this way, it is possible to first run a number of COAs, and then later compare the effects of the COAs to find the best possible COA for the plan.

All interactions with the CGI scripts are done by following links from web pages or by submitting HTML forms. Therefore, the operational planner does not need to know anything about the CGI scripts; they are hidden behind the web browser.

8.4.2 Controlling the TINL CPN Simulator from a Web Environment

In this section we describe how the TINL CPN simulator can be controlled from a web environment. We describe how the parameters for the model are retrieved from the web browser, how the simulator is initialised, and how the results from simulating the TINL are produced and shown in the web browser. The control is completely hidden from the operational planner behind the web-based interface. The technical details of the method which is used for controlling

the CPN simulator from a web browser is described in detail in [54].

The simulator is controlled automatically by a batch script when the CGI script is executed. The source code for the batch script is not included here. Instead, we will discuss the elements of the batch script at a conceptual level. Below we first show the sequence of actions in the batch script used for controlling the TINL CPN simulator when the CGI script is invoked from an HTML form. Afterwards, we discuss more details on the actions of the batch script.

i. Retrieve the model name and delay name from the HTML form.

ii. Load a TINL-specification file and a delay file.

iii. Retrieve the TINL specific parameters from the HTML form.

iv. Create output directories.

v. Save an HTML document with input parameters.

vi. Calculate initial marking.

vii. Run simulation and collect data.

viii. Generate and save graphics using gnuplot.

ix. Produce and save HTML documents containing results for later use.

x. Send the summary HTML document to the web browser.

xi. Quit the CGI script.

When the CGI script is invoked, the batch script retrieves the parameters from the HTML form. However, the parameters are dependent on the given TINL. Therefore, it first retrieves the model name and the delay name to know in which directory the TINL specification file and the delay file can be found. Based on the information in the TINL specification file, the remaining parameters can be retrieved from the HTML form.

The next action is to prepare for producing output while simulating the model. Therefore the necessary output directories are created, and an HTML document (`run-cpn.html`) equal to the one used for setting parameters for this simulation – but including the values typed in by the operational planner – is saved to store all initial parameters for this specific delay file and COA. This document can be used to run a similar COA with a few changed parameters. This makes it easy to specify several similar COAs.

The parameters of the simulator are then initialised with the values from the TINL specification file, the delay file, and the HTML form. Afterwards, the simulator is ready to be started. While simulating, data will be collected and stored in the output directory.

After the simulation has stopped, the data which has been collected during the simulation is used for generating graphs. The graphs are created by first generating a file with a script for the graph plotting tool gnuplot [32], and then gnuplot is executed with the script as parameter. A graph (`res<nodeno>.png`)

is created for each of the nodes which the operational planner has marked as observable in the input HTML form. These graphs are used to give a very detailed profile of each individual node. Another graph (`comp.png`) which contains graphs from all observable nodes is also generated. It is used to compare the profiles of the observable nodes with each other. An example of such a graph was given in Fig. 8.6 in Sect. 8.2.

Next, the file `detailedindex.html`, containing an HTML document including the graphs of the individual observable nodes, is generated and saved. Finally, an HTML document (`index.html`) is generated to give an overview and access to all the HTML documents and simulation results. This document is saved in a file, but is also sent to the web browser to give the SME immediately access to an overview of the simulation results.

8.4.3 Why Choosing a Web-Based Simulation Environment

A web environment is useful for this project due to the fact that the SMEs can easily access the simulation application from anywhere. From the modeller's point of view it is also easy to create a domain-specific GUI using standard web techniques.

Using HTML forms and CGI scripts for controlling the simulation environment has turned out to be a good choice for this application. First of all, because it requires only a few basic programs like the interpreter Perl [109] and the Standard ML runtime environment to be installed on the web-server. Secondly, the CGI script containing the TINL CPN simulator code is generated completely automatically from Design/CPN, and is only generated once when installing the web-site.

The most notable disadvantage is that the TINL CPN simulator has a size of about 12 MB, and the memory image which takes up the memory while executing the CGI script is also rather large. However, one of the assumptions of the web environment is that only a few SMEs are using the CGI scripts at the same time. Therefore, the relatively large memory requirement of the web server is not a critical problem. If it turns out to be a problem in practice, then the TINL CPN simulator of the CGI scripts may be distributed to other machines.

8.5 Performance Results on Execution Time

In this section we present results on the time-effectivity[2] of the developed TINL CPN simulators. As mentioned previously, the time it takes to apply the method is relevant because the generic CPN model has been developed to be able to do faster analysis of TINLs than previous CPN models. We compare the effectivity of the generic CPN model presented in Sect. 8.3 with the fixed CPN models obtained using the method presented in [108]. In addition, the generic CPN model has been simulated using both the old simulator and the new simulator [77] of Design/CPN. The reason for using the new simulator of

[2]We have used a Sun 4, 128 MB RAM, sparc-solaris 5.8.

#TINL Nodes	Model	Simulator	Draw CP-net	Switch	Simulate	Total Time
5	Fixed	Old	21,390	86,000	100	107,490
	Generic	Old	–	–	140	140
	Generic	New	–	–	20	20
30	Fixed	Old	24,860	396,000	390	421,250
	Generic	Old	–	–	320,440	320,440
	Generic	New	–	–	520	520

Table 8.1: Execution time in milliseconds for different models using different simulators.

Design/CPN is that it turned out that the old simulator was too slow for the folded CPN model. Table 1 contains execution time in milliseconds for two TINLs with 5 and 30 nodes. A 30 node TINL can be considered as a TINL with a real-world size.

For the fixed CPN models there are extra time penalties for drawing the actual CPN model which are not present for the generic CPN model. The table shows that it takes more than 20 seconds to draw the full CPN model. In addition, extra time is needed for generating simulation code (switch) for the CPN model. The switch time depends heavily on the number of nodes in the TINL. The 5-node TINL takes about 86 seconds to switch, while the 30-node TINL takes about 396 seconds. The simulation time for the fixed models is very low. The reason is that the old simulator code is quite effective for CP-nets with few tokens on each place, which is the case for the fixed CPN models. The overall time it takes to draw, switch, and simulate the CP-net of a TINL using the fixed CPN model is relatively high: more than 100 seconds for a small TINL with only 5 nodes.

The generic CPN model avoids the need of drawing and switching a CPN model each time a new TINL is to be analysed. The switch is made only once for the generic model, and the simulator can be applied to any TINL afterwards. Therefore, the switching time is not included in the table. Simulating the generic CPN model using the old simulator is comparable to simulating the fixed model for very small TINLs. However, as the simulation time of the 30-node TINL shows, it takes longer time to simulate the generic CPN model for TINLs with 30 nodes. The reason is that the old simulator is ineffective when simulating CPN models with many tokens on a place. When considering the total time, the generic CPN model using the old simulator is almost as inefficient as using the fixed CPN model.

The large simulation time of the generic CPN model has been solved using the new CPN simulator which is optimised for simulations with many tokens on a place. The improvement can be seen from the fact that the simulation time of the generic CPN model using the new simulator is close to the simulation time of the fixed CPN model using the old simulator. The simulation time for the generic CPN model includes the first step where the data is loaded into the

model and tokens are generated. However, for the 30-node TINL specification file the first step takes only about 50 milliseconds. In conclusion we note that the simulation time using the new simulator and the generic CPN model is fully acceptable for web simulations.

8.6 Conclusion

In this paper we have presented a web-site which contains a CPN simulator for effect based operational planning. The CPN model is used to evaluate alternative COAs with respect to timing and probability profiles. The input for the model consists of a complete influence net, a timing profile for actionable events, and estimated probabilities for the events in the influence net. The results of simulating the CPN model are, e.g. graphs displaying expected probabilities for each of the observable nodes at different points in time. This information can be used in the process of timing an operation by selecting the best of the examined COAs.

Future work may include embedding the TINL CPN simulator into the CAT tool as a component. The simulation results could then be displayed within the CAT tool, and thereby lead to a more iterative evolution of TINLs. It would probably be more effective to apply the method if, when adding an extra node to a TINL, one could see the consequences on the simulation results immediately after adding the node.

The IEEE 1516 standard called High Level Architecture (HLA) [93] can be used for interconnecting simulation models to become so-called federations of simulation models. By creating an interface for the HLA runtime infrastructure to the TINL CPN simulator, it would be possible to use output from simulations of e.g. a war game as input to the simulator. There exist simulation models at different strategic levels of war planning, and the outcome of using the TINL simulator in such a federation could be a method to easily determine the effect of alternative COAs based on these other simulation models.

Acknowledgements. This research was conducted in cooperation between University of Aarhus and the C3I Center of George Mason University, with partial support provided by the U.S. Office of Naval Research under Grant No. N00014-01-1-0538.

Chapter 9

Towards a Monitoring Framework for Discrete-Event System Simulations

The paper presented in this chapter will be published in the proceedings of the 6th International Workshop on Discrete-Event Systems 2002 (WODES'02).

[62] B. Lindstrøm and L. Wells. Towards a Monitoring Framework for Discrete-Event System Simulations. *To appear in Proceeding of Workshop on Discrete Event Systems*, October 2002.

Except for minor typographical changes the content of this chapter is equal to the original version of the paper in [62].

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.
Key words: Discrete-event system simulation, coloured Petri nets, computer tools.

9.1 Introduction

A variety of formalisms, e.g. finite-state machines [38], statecharts [36], and Petri nets [88], 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 [40] (MSC), or data collection for performance analysis.

Based on our experiences with implementing and using Design/CPN [27] which is a tool for coloured Petri nets (CP-nets or CPN) [44, 45], we have observed that the design and implementation of efficient and effective tool support

*Department of Computer Science, University of Aarhus, Denmark.
E-mail: {blind,wells}@daimi.au.dk.

for a specific formalism is generally focused on the formalism, while extracting information for other purposes is typically done using ad hoc methods. That 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 [58] 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 9.2 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 9.3 presents the general monitoring framework for simulation models of discrete-event systems. Section 9.4 describes how the monitors presented in Sect. 9.2 can be realised using the general framework presented in Sect. 9.3. Finally, Sect. 9.5 concludes and gives directions for future work.

9.2 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 [45] which contains a detailed description of both the model and timed coloured Petri nets.

9.2.1 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 9.1 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

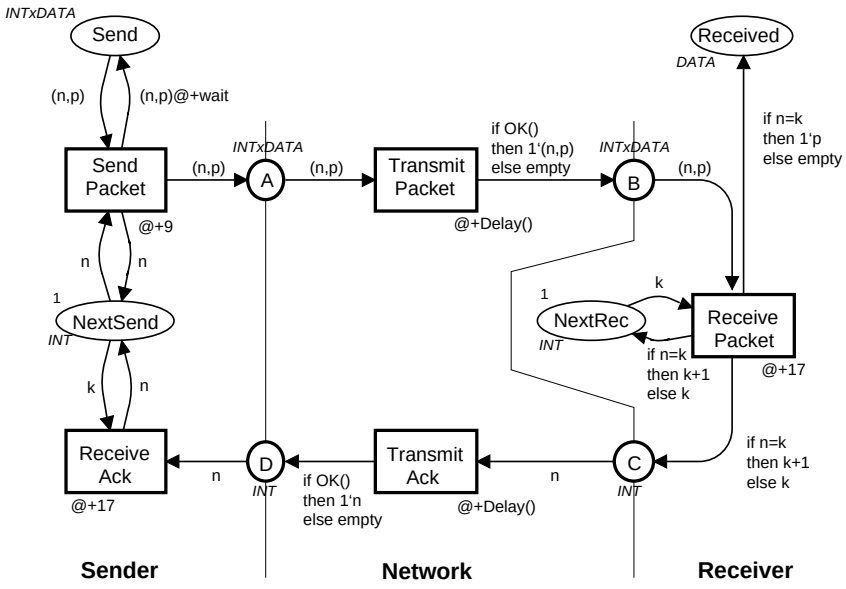


Figure 9.1: Timed communication protocol.

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.

9.2.2 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 behaviour of the system during a simulation, e.g. by updating message sequence charts or domain-specific graphics. *Performance monitors* measure and report

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

Figure 9.2: Excerpt from the log file for the receiver.

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 9.2 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 9.3 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

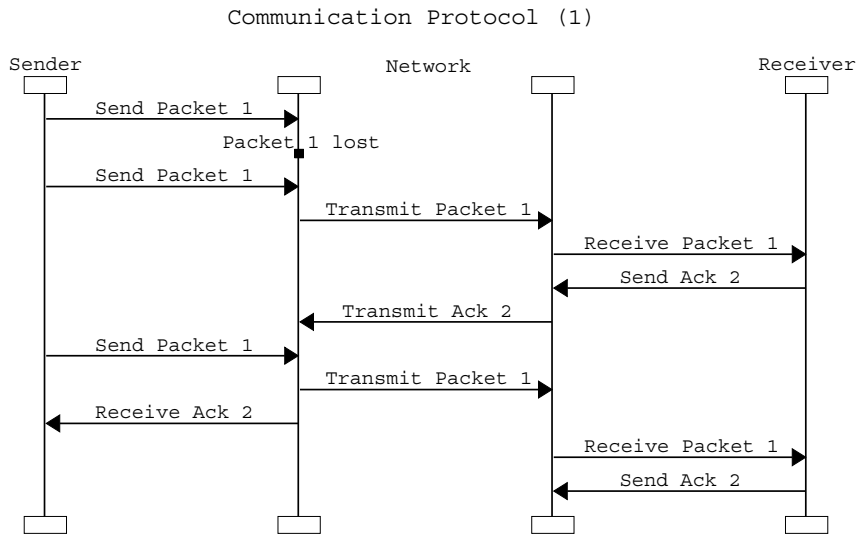


Figure 9.3: MSC for communication protocol.

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.

9.3 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. 9.3.1 we give a description of the essential parts of a monitor. Section 9.3.2 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.

9.3.1 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. 9.2 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:

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

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

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

As an example, consider the log-file monitor in Sect. 9.2.2. 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

```
if check (event, state)                              1
    then act (observe (event, state), state)         2
```

Figure 9.4: Relation between monitor functions.

to the file. The action updates the file with the observed string. The state of the model is left unchanged.

Figure 9.4 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. 9.4 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. 9.4 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. 9.4 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.

9.3.2 Interface of Monitors

In this section we provide a detailed specification for the interface of a monitor. Figure 9.5 shows the interface of a monitor, which is specified as a Standard ML [82] 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, 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.

```
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 9.5: Interface of a 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. 9.4. 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`.

9.4 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. 9.3. We will also discuss the advantages of providing support for both standard and user-defined monitors.

9.4.1 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

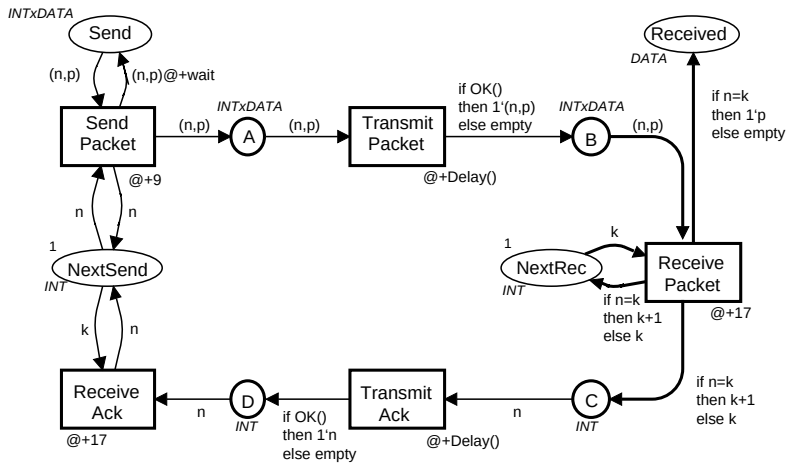


Figure 9.6: Determining events and states to monitor.

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. 9.2.2 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. 9.6 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. 9.5. 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.

Defining the log-file monitor is fairly easy in Design/CPN. The complete code for the log-file monitor can be seen in Fig. 9.7, and a description of how the

```
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 9.7: Code for log-file monitor.

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. 9.7. 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. 9.7) in order to generate a file that resembles the excerpt shown in Fig. 9.2.

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. 9.7. 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.

9.4.2 Standard and User-Defined Monitors

Based on the monitoring framework presented in Sect. 9.3, 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. 9.2. 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. 9.5, 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.

9.5 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 10

Annotating Coloured Petri Nets

The paper presented in this chapter will be published in the proceedings of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'02).

[61] B. Lindstrøm and L. Wells. Annotating Coloured Petri Nets. To appear in proceedings of K. Jensen, editor, *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Department of Computer Science, August 2002. Online: http://www.daimi.au.dk/CPnets/workshop02/.

Except for minor typographical changes the content of this chapter is equal to the original version of the paper in [61].

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.

10.1 Introduction

Coloured Petri nets (CP-nets or CPNs) were formulated by Kurt Jensen [44, 45] with the primary purpose of specifying, designing, and analysing concurrent systems. The tools Design/CPN [17, 27] and CPN Tools [23] 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 [46] have identified the need for additional facilities in the tools.

One industrial project described in [10] 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 [58] 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 [62].

*Department of Computer Science, University of Aarhus, Denmark.
E-mail: {blind,wells}@daimi.au.dk.

Other projects have shown that visualisation of behaviour using so-called message sequence charts (MSCs) [40] is very useful in combination with CP-nets. As a consequence, a library [71] has been developed for creating MSCs during simulations. Other similar libraries are Mimic [87], which is used for visualisation, and Comms/CPN [30], 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 [44]. 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

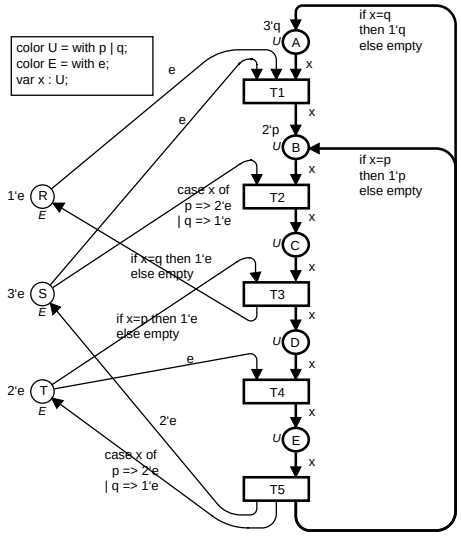


Figure 10.1: The basic CP-net for the resource allocation system.

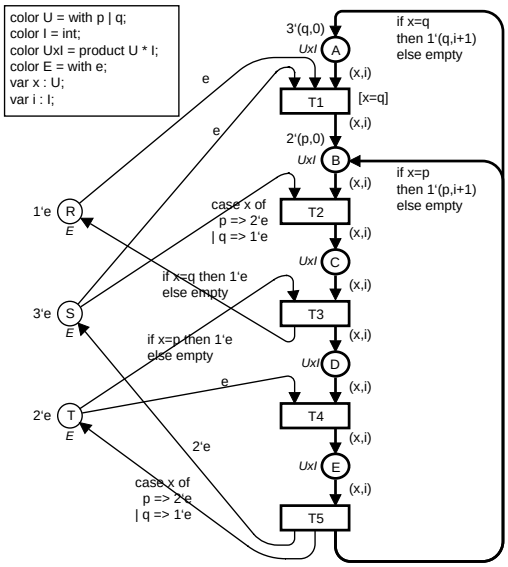


Figure 10.2: The CP-net for the resource allocation system extended with a counter.

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 10.2 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. 10.3 we informally introduce our proposal for how to annotate CP-nets. Section 10.4 discusses how multiple annotation layers can be used for visualisation using MSCs. In Sect. 10.5 we give the formal definitions for annotating CP-nets. Finally, in Sect. 10.6 we conclude and give directions for future work.

10.2 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 [44, 45]. At least three variations of the resource allocation CP-net can be found in these volumes: a *basic* version (shown in Fig. 10.1) suitable for full state space analysis; an *extended* version (shown in Fig. 10.2) 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. 10.1 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. 10.2 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 [45] 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. 10.2 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

[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.

removed by manually modifying the colour sets and arc inscriptions.

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.

10.3 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 10.3.1 gives an informal introduction to annotations and annotation layers. Section 10.3.2 describes the intuition of how to translate an annotated CP-net to a matching CP-net. Section 10.3.3 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. 10.5.

10.3.1 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. 10.2 can be added as annotations. Recall that Fig. 10.2 shows how the CP-net from Fig. 10.1 can be modified to include the cycle counters as part of the token colours.

Figure 10.3 contains an annotated CP-net for the basic CP-net for the resource allocation system from Fig. 10.1. In Fig. 10.3 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.

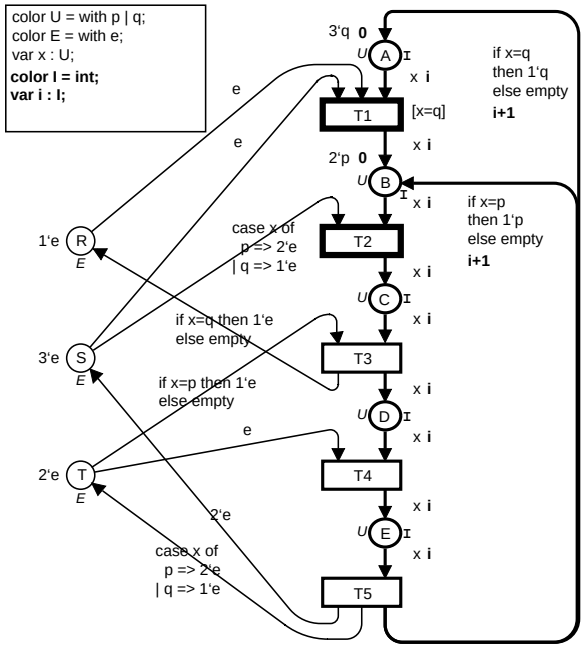


Figure 10.3: Annotated CP-net for the resource allocation system.

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. 10.3, 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. 10.5.2.

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 10.5.2 and 10.5.3 discuss the restrictions about which kinds of auxiliary arc expressions are allowed.

10.3.2 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. 10.3) and the underlying CP-net (referred to as CPN and shown in Fig. 10.1) 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. 10.4). The general rules for translating an arbitrary annotation layer and its underlying CP-net are presented in Sect. 10.5.3. In the following, we shall say that a place/arc is annotated/non-annotated in a matching CP-net (like Fig. 10.4) if it is annotated/non-annotated in the corresponding annotated CP-net (like Fig. 10.3).

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 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

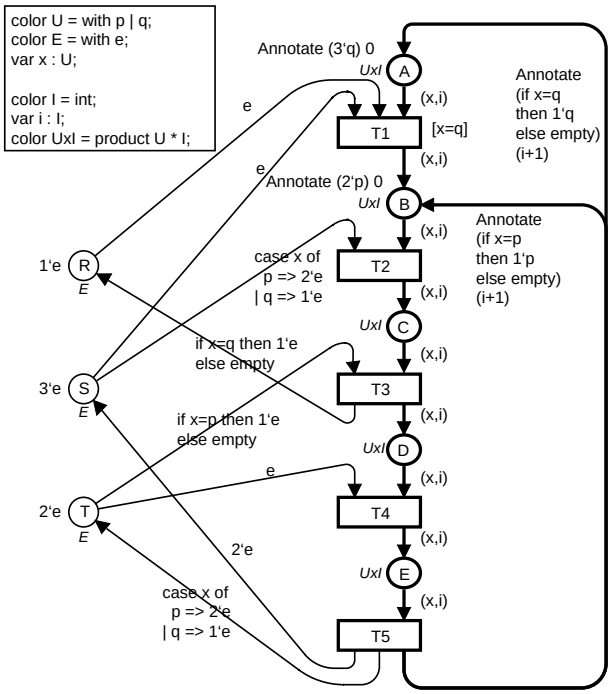


Figure 10.4: Matching CP-net for the resource allocation system.

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. 10.4 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.

10.3.3 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 10.5 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. 10.3 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. 10.1, 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. 10.5 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. 10.5.4.

10.4 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. 10.1 in Sect. 10.2. The purpose of this section is to illustrate that multiple annotation layers can be

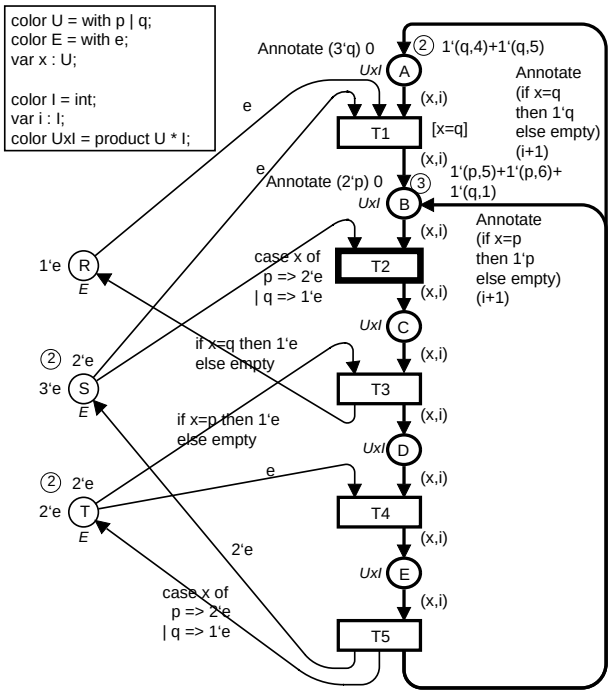


Figure 10.5: Marking of the matching CP-net for the resource allocation system.

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 10.6 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. 10.6 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) 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

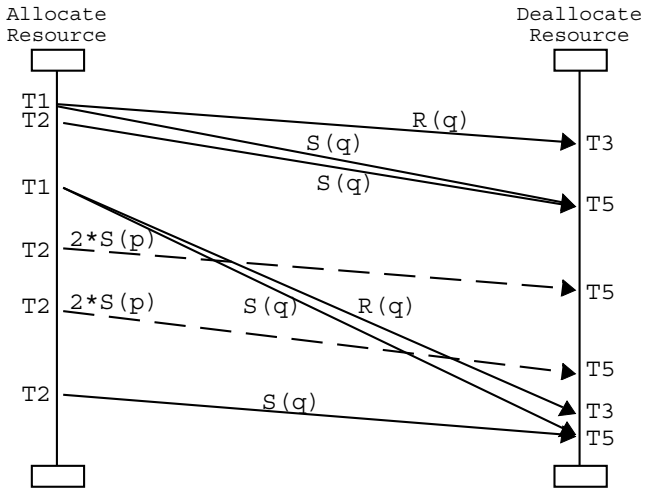


Figure 10.6: Message sequence chart for the resource allocation system.

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 10.7 depicts how the basic CP-net for the resource allocation system can be annotated to generate the MSC in Fig. 10.6. 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.

The annotation layer in Fig. 10.3 from Sect. 10.3.1 adds a cycle counter to the CP-net. The value of the cycle counter could be included on the arrows in Fig. 10.6 in addition to the type of resource and process. To obtain this, an annotation layer that resembles the MSC Annotation Layer in Fig. 10.7 can

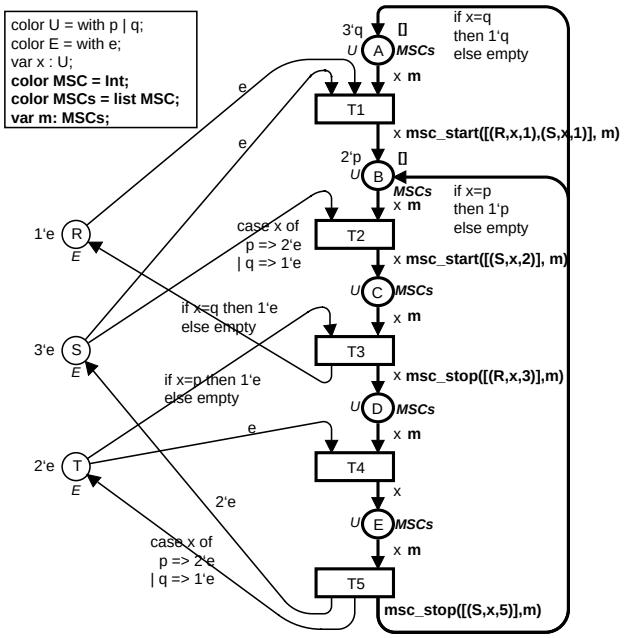


Figure 10.7: Resource allocation system with MSC annotation layer.

be added on top of the cycle counter annotation layer in Fig. 10.3. 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 10.8 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.

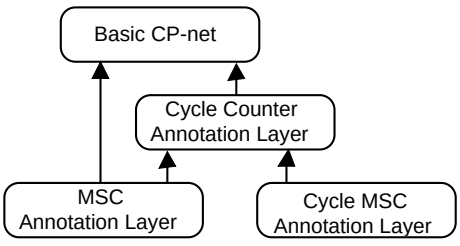


Figure 10.8: Structure of annotation layers for a basic CP-net.

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.

10.5 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 10.1: Let CPN=(Σ, P, T, A, N, C, G, E, I) be a CP-net as defined in Def. 2.5 in [44]. 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 10.5.1 introduces terminology regarding multi-sets of annotated colours. Section 10.5.2 defines annotation layers by describing the auxiliary net inscriptions that are allowed in the annotation layer. Section 10.5.3 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 10.5.4 defines matching behaviour. Finally, Sect. 10.5.5 discusses the use of multiple annotation layers.

10.5.1 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 10.1 *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* $s \in S$ *and* $a \in AN$. $\pi((s,a))=s$ *is the projection of the annotated element (s,a) onto the non-annotated element s.*

A multi-set of annotated elements over $S \times AN$ *is a multi-set over* $S \times AN$.

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* $\mathtt{am}_\pi(s) = (\sum_{a \in AN} \mathtt{am}(s,a))\grave{}s$. $\pi(\mathtt{am})$ *is the projection of* `am` *onto the non-annotated multi-set determined by* `am`.

If `am` *is multi-set over* $S \times AN$, `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. 10.3.2 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 10.2 *Given an annotation* $\mathtt{a} \in AN$ *and a multi-set* $\mathtt{m}=\sum_{s \in S} \mathtt{m}(s)\grave{}s$ *over a set S, the function* `Annotate` *is defined to be:*

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

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. 10.1 and 10.2, $\pi(\mathtt{Annotate\ m\ a}) = \mathtt{m}$, for all multi-sets `m` and all annotations `a`.

10.5.2 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. 10.3.2 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 [44]

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 10.3 *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:*

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

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

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

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 $\mathcal{A}$.

10.5.3 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 10.4 *Let $(CPN, \mathcal{A})$ be an annotated CP-net, where $CPN=(\Sigma, P, T, A, N, C, G, E, I)$ 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 $CPN^*=(\Sigma^*, P^*, T^*, A^*, N^*, C^*, G^*, E^*, I^*)$ 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. $E^*(a) = \begin{cases} E(a) & \text{if } a \notin A_{\mathcal{A}} \\ \textit{Annotate } E(a)\ E_{\mathcal{A}}(a) & \text{if } a \in A_{\mathcal{A}} \end{cases}$

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

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 $(C(p(a))\times Type(E_{\mathcal{A}}(p(a))))_{MS}$ which is exactly $(C(p(a))\times C_{\mathcal{A}}(p(a)))_{MS}$, 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 $(C(p(a))\times Type(I_{\mathcal{A}}(p(a))))_{MS}$ which is exactly $(C(p(a))\times C_{\mathcal{A}}(p(a)))_{MS}$, as required. $I^*(p)$ is a closed expression for all p, since I(p) is a closed expression for all p, and $I_{\mathcal{A}}(p)$ is closed for all $p\in P_{\mathcal{A}}$. 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 10.5 *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)\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:

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

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

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

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

Sound Annotation Layers

Definition 10.3 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 10.6 *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 10.6 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. 10.6. 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).

10.5.4 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 10.1 *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 10.1 can be found in Appendix 10.A.

10.5.5 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 10.7 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 10.7 *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. 10.5.3. Then CPN* with multiple annotation layers is defined by:*

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

10.6 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 [50], 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 disable different kinds of instrumentation when analysing different aspects of one particular model.

Ex*Spect* [1] 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. 10.1 in Sect. 10.2. 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.

10.A Proof of Matching Behaviour

Theorem 10.1 (same as in Sect. 10.5.4) *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 [44]. 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:

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

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

$$\pi\big(E^*(a)\langle b^*\rangle\big) \overset{Def.}{=} \overset{10.4.viii}{\pi}\big((\text{Annotate } E(a)\ E_{\mathcal{A}}(a))\langle b^*\rangle\big) \overset{Defs.}{=} \overset{10.1\&10.2}{E(a)\langle b^*\rangle} \overset{Def.}{=} \overset{10.5.ii}{E(a)\langle\pi(b^*)\rangle}$$

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 [44] 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. 10.4.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. 10.1) 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:

$$\pi(TE^*)=\{(p, c) \mid p \in P \text{ and } c \in C(p)\} \overset{Def.}{=} \overset{2.7in\ [44]}{TE}$$

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:

$$\forall p \notin P_{\mathcal{A}}\text{: } (\pi(M_0^*))(p) \overset{Def.\ 10.5.i}{=} M_0^*(p) = I^*(p) \overset{Def.\ 10.4.ix}{=} I(p) = M_0(p)$$

Consider annotated places:

$$\begin{aligned}\forall p \in P_{\mathcal{A}}\text{: } (\pi(M_0^*))(p) &\overset{Def.\ 10.5.i}{=} \pi(M_0^*(p)) \\ &= \pi(I^*(p)) \\ &\overset{Def.\ 10.4.ix}{=} \pi(\text{Annotate } I(p)\ I_{\mathcal{A}}(p)) \\ &\overset{Defs.\ 10.1\&10.2}{=} I(p) \\ &= M_0(p)\end{aligned}$$

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 [44] 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. 10.5.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. 10.4.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 [44]), 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 [44], 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 [44]) we have that:

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

Consider non-annotated places and non-annotated arcs. Since $E^*=E$ for all non-annotated arcs (by Def. 10.4.viii), and $M_1^*=\pi(M_1^*)$ for all non-annotated places (by Def. 10.5.i), it follows from $(*)$ that:

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

which by the fact that $\pi(b^*)=b^*$ for all variables in Var(E(a)) (by Def. 10.5.ii) and the definition of $\pi(Y^*)$ (Def. 10.5.iii) is equivalent to:

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

Consider annotated places and annotated arcs. From Def. 10.1 and $(*)$, it follows that:

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

which by Defs. 10.1 and 10.5.i is equivalent to:

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

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

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

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. 10.6 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 [44]) gives us that:

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

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

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

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. 10.5.i). For all non-annotated arcs $E^*=E$ (from Def. 10.4.viii). It follows from these facts and ($\diamond\diamond$) that:

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

which by the fact that $\pi(b^*)=b^*$ for all variables in $Var(E(a))$ (by Def. 10.5.ii) and the fact that $\pi(Y^*)=Y$ is equivalent to:

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

We will show that $\pi(M_2^*)=M_2$ for annotated places. From the definition of π for multi-sets and markings (Defs. 10.1 and 10.5.i) and from ($\diamond\diamond$), it follows that:

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

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

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

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:

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

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

List of Figures

List of Tables

Bibliography

[1] W. v. d. Aalst, P. d. Crom, R. Goverde, K. v. Hee, W. Hofmann, H. Reijers, and R. v. d. Toorn. ExSpect 6.4 – An Executable Specification Tool for Hierarchical Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 455–464. Springer-Verlag, 2000.

[2] 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.

[3] T. Andersen. Improved Methodology for the Design of Communication Protocols in Security Systems. In *Technical Report*. Dalcotech A/S, Denmark, May 1996.

[4] P. Athena. *PROTOS User Manual.* Pallas Athena BV, Plasmolen, The Nederlands, 1997.

[5] 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.

[6] 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.

[7] A. Bobbio, A. Puliafito, M. Scarpa, and M. Telek. WebSPN: A WEB-accessible Petri Net Tool. In *Proceedings of International Conference on Web-Based Modeling and Simulation*, San Diego, CA, January 1998.

[8] A. Campos and D. Hill. Web-Based Simulation of Agent Behaviours. Technical report, ISIMA, Computer Science & Modelling Institute, Blaise Pascal University, France, 1998.

[9] C. Caoellmann 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*, Zaragoza, Spain, 1994.

[10] CAPLAN Project, Online: http://www.daimi.au.dk/CPnets/CAPLAN/.

[11] CCITT. Specification and Description Language SDL, Recommendation Z100-Z104,. Technical report, ITU, 1992.

[12] K. Chang, P. Lehner, A. Levis, A. Zaidi, and X. Zhao. On Causal Influence Logic. Technical report, Center of Excellence for C3I, George Mason University, 1994.

[13] B. Cheikes and A. Gertner. Software Instrumentation for Intelligent Embedded Training. The MITRE Corporation, Bedford, MA, USA. http://www.mitre.org/support/papers/.

[14] 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.

[15] S. Christensen and N. Hansen, editors. *Coloured Petri Nets Extended with Channels for Synchronous Communication*, Daimi PB - 390, April 1992.

[16] S. Christensen and J. Jørgensen. Analysis of Bang & Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*. Springer-Verlag, 1997.

[17] S. Christensen, J. Jørgensen, and L. Kristensen. Design/CPN – A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217, pages 209–223. Springer-Verlag, 1997.

[18] S. Christensen and L. Kristensen. State Space Analysis of Hierarchical Coloured Petri Nets. In B. Farwer, D. Moldt, and M. Stehr, editors, *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation, Hamburg, Germany*, volume 205, pages 32–43. University Hamburg, Fachberich Informatik, 1997.

[19] S. Christensen and K. Mortensen. Parameterisation of Coloured Petri Nets. In *Technical Report DAIMI PB-521*. Department of Computer Science, University of Aarhus, Denmark, March 1997.

[20] H. Clausen and P. Jensen. Validating and Performance Analysis of Network Algorithms by Coloured Petri Nets. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models*, pages 280–289. IEEE Computer Society Press, 1993.

[21] R. Clemen. *Making Hard Decisions: An Introduction to Decision Analysis*. Duxbury Press, 1996. 2nd edition.

[22] D. Comer. *Internetworking with TCP/IP*, volume 1. Prentice Hall, 1995.

[23] CPN Tools, Online: http://www.daimi.au.dk/CPnets/CPN2000/.

[24] J. C. A. de Figueiredo, B. Lindstrøm, L. M. Kristensen, J. Bogorad, S. Christensen, K. Jensen, K. H. Mortensen, J. S. Thomasen, and L. M. Wells. *The Desktop ClassRoom*. HP-CPN Project Report Series HP-CPN-3. HP-CPN Centre, Department of Computer Science, University of Aarhus, May 1999.

[25] Deloitte and T. Bakkenist. *ExSpect*. Product Management ExSpect, P.O. Box 23103, 1100 DP Amsterdam, The Nederlands. http://www.exspect.com.

[26] Deloitte and T. Bakkenist. *ExSpect User Manual*. Product Management ExSpect, P.O. Box 23103, 1100 DP Amsterdam, The Nederlands.

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

[28] Sun, Online: http://www.exspect.com.

[29] M. Fowler and K. Scott. *UML Distilled – A Brief Guide to the Standard Object Modelling Language*. Addison-Wesley, second edition, 1999.

[30] G. Gallasch and L. 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–544*, pages 79–93. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/CPnets/workshop01/.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[32] Gnuplot, Online: http://www.cs.dartmouth.edu/gnuplot_info.html.

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

[34] M. Gries, J. Janneck, and M. Naedele. Reusing Design Experience for Petri Nets Through Patterns. In *Proceedings of High Performance Computing '99*, pages 453–458, 1999.

[35] S. Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

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

[37] P. G. Hoel, S. C. Port, and C. J. Stone. *Introduction to Probability Theory*. Houghton Mifflin, 1971.

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

[39] A. Horváth, A. Pufliafito, M. Scarpa, M. Telek, and O. Tomarchio. Design and Implementation of a WEB-based non-Markovian Stochastic Petri Net Tool. Technical report, University of Catania, 95125 Catania, Italy, 1998.

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

[41] F. J. and E. Dimitrov. Verification of SDL Protocol Specifications using Extended Petri Nets. In J. Billington and M. Diaz, editors, *Workshop on Petri Nets and Protocols of the 16th International Conference on Application and Theory of Petri Nets*, pages 1–12, 1995.

[42] Java, Online: http://developer.java.sun.com/.

[43] J. Jensen. *An Introduction to Bayesian Networks.* UCL Press, 1996.

[44] 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.

[45] 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.

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

[47] K. Jensen, S. Christensen, and L. M. Kristensen. *Design/CPN Occurrence Graph Manual.* Department of Computer Science, University of Aarhus, Denmark, 1996. Online: http://www.daimi.au.dk/designCPN/man/.

[48] L. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, December 1998.

[49] 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.* HP-CPN Project Report Series HP-CPN-1. HP-CPN Centre, Department of Computer Science, University of Aarhus, May 1998.

[50] C. Lakos. On the Abstraction of Coloured Petri Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 42–61. Springer-Verlag, 1997.

[51] C. Lakos. Composing Abstraction of Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 323–345. Springer-Verlag, 2000.

[52] A. Levis. Course of Action Development for Information Operations. In *Phalanx*, volume 33, No. 4. Military Operations Research Society, 2000.

[53] B. Lindstrøm. Web-Based Interfaces for Simulators of Coloured Petri Net Models. In K. Jensen, editor, *Workshop on the Practical Use of High-Level Petri Nets, DAIMI PB–547*, pages 15–33. University of Aarhus, Department of Computer Science, 2000. Online: http://www.daimi.au.dk/pn2000/proceedings.

[54] B. Lindstrøm. Web-Based Interfaces for Simulators of Coloured Petri Net Models. *International Journal on Software Tools for Technology Transfer*, 3(4):405–416, September 2001.

[55] B. Lindstrøm and S. Haider. Equivalent Coloured Petri Net Models of a Class of Timed Influence Nets with Logic. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB–544*, pages 35–54. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/CPnets/workshop01/.

[56] B. Lindstrøm and L. Wagenhals. Operational Planning using Web-Based Interfaces to a Coloured Petri Net Simulator of Influence Nets. In C. Lakos, R. Esser, L. Kristensen, and J. Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 115–124. Australian Computer Society Inc., June 2002.

[57] B. Lindstrøm and L. Wells. Batch Scripting Facilities for Design/CPN. In K. Jensen, editor, *Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, DAIMI PB–541*, pages 79–97. University of Aarhus, Department of Computer Science, 1999. Online: http://www.daimi.au.dk/CPnets/workshop99/.

[58] 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/.

[59] B. Lindstrøm and L. Wells. Performance Analysis using Coloured Petri Nets. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, May 1999.

[60] B. Lindstrøm and L. Wells. *Tool Support for Simulation Based Performance Analysis using Coloured Petri Nets*, 1999. Department of Computer Science, University of Aarhus, Denmark.

[61] B. Lindstrøm and L. Wells. Annotating Coloured Petri Nets. In K. Jensen, editor, *To Appear in Proceedings of Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Department of Computer Science, August 2002. Online: http://www.daimi.au.dk/CPnets/workshop02/.

[62] B. Lindstrøm and L. Wells. Towards a Monitoring Framework for Discrete-Event System Simulations. In *To appear in Proceeding of Workshop on Discrete Event Systems*. IEEE, October 2002.

[63] B. Lindstrøm, L. M. Wells, J. Bogorad, S. Christensen, K. Jensen, L. M. Kristensen, K. H. Mortensen, and J. S. Thomasen. *HTTP Web Servers – Part B*. HP-CPN Project Report Series HP-CPN-2. HP-CPN Centre, Department of Computer Science, University of Aarhus, November 1998.

[64] L. Lorentsen and L. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System using Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 346–366. Springer-Verlag, 2000.

[65] L. Lorentsen and L. Kristensen. Exploiting Stabilizers and Parallelism in State Space Generation with the Symmetry Method. In *Proceedings of the Second Internationel conference on Application of Concurrency to System Design*, pages 211–220. IEEE, 2001.

[66] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling Feature Interaction Patterns in Nokia Mobile Phones using Coloured Petri Nets and Design/CPN. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB–544*. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/CPnets/workshop01/.

[67] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling Feature Interactions in Mobile Phones. In *Feature Interaction in Composed Systems*. ECOOP 2001 Workshop, June 2001.

[68] O. Madsen, B. Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[69] T. Mailund. Parameterised Coloured Petri Nets. In K. Jensen, editor, *Proceedings of Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 133–151. Department of Computer Science, University of Aarhus, Denmark, 1999.

[70] MARS-Team. CPN-AMI Home page. http://www-src.lip6.fr/cpn-ami.

[71] Message Sequence Charts in Design/CPN, Online: http://www.daimi.au.dk/designCPN/libs/mscharts/.

[72] Meta Software Corporation, 125 Cambridge Park Drive, Cambridge MA 02140, USA. *Work Flow Analysis. User's Manual*, 1994.

[73] J. Meyer, A. Movaghar, and W. Sanders. Stochastic Activity Networks: Structure, Behavior, and Application. In *Proceedings of International Conference on Timed Petri Nets*, pages 106–115, Torino, Italy, July 1985.

[74] Microsoft. Online: http://www.microsoft.com.

[75] J. Miller, A. Seila, and X. Xiang. The JSIM Web-Based Simulation Environment. Technical report, University of Georgia, Computer Science Department, 415 GSRC, Athens, GA, 30602-7404, 1999.

[76] MITRE. Online: http://www.mitre.org.

[77] K. 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, DAIMI PB–544*, pages 57–74. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/CPnets/workshop01/.

[78] K. Mortensen and V. Pinci. Modelling the Work Flow of a Nuclear Waste Management Program. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *LNCS*, pages 376–395. Springer-Verlag, June 1994.

[79] A. Movaghar and J. Meyer. Performability Modeling with Stochastic Activity Networks. In *Proceedings of Real-Time Systems Symposium 1984*, Austin, TX, December 1984.

[80] M. Naedele and J. Janneck. Design Patterns in Petri Net System Modeling. In *Proceedings of 4th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 47–54, Monterey, CA, August 1998.

[81] S. Nimsgern and F. Vernet. Communication between Coloured Petri Net Simulations and External Processes. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 2000.

[82] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[83] L. C. Paulson. *ML for the Working Programmer, 2nd edition*. Cambridge University Press, July 1996.

[84] V. Pinci and R. Shapiro. An Integrated Software Development Methodology Based on Hierarchical Coloured Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 524 of *LNCS*, pages 227–252, 1991.

[85] D. Poitrenaud and D. Prun. *CPN/DESIR version 1.0 – User Guide*. MASI Lab, Blaise Pascal Institute, University P. & M. Curie, Paris, France.

[86] J. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In J. Billington and W. Reisig, editors, *Proceedings of the 17th International Petri Net Conference*, volume 1091 of *LNCS*, pages 400–419. Springer-Verlag, 1996.

[87] 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/.

[88] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[89] Renew. Online: http://www.renew.de.

[90] J. Rosen and W. Smith. Influence Net Modelling with Causal Strengths: an Evolutionary Approach. In *Proceedings of Command and Control Research and Technology Symposium*, pages 699–708. Naval Post Graduate School, Monterey, CA, USA, 1996.

[91] W. H. Sanders. *Construction and Solution of Performability Models Based on Stochastic Activity Networks*. PhD thesis, University of Michigan, USA, 1988.

[92] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley and Sons Ltd., 2000.

[93] Simulation Interoperability Standards Organisation (SISO), http://www.sisostds.org/stdsdev/hla/. *HLA Standards Development*.

[94] B. Stroustrup. *The C++ Programming Languages*. Addison-Wesley, 2000.

[95] Sun, Online: http://developer.java.sun.com/.

[96] C. Szyperski and C. Pfister. Component-Oriented Programming. In M. Muehlhaeuser, editor, *WCOP'96 Workshop Report*, Special Issues in Object Oriented Programming. dpunkt Verlag, 1997.

[97] UltraSAN.
Online: http://chaos.crhc.uiuc.edu/UltraSAN/UltraSAN.html.

[98] *UltraSAN User's Manual: Version 3.0*. Center for Reliable and High-Performance Computing Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1995.

[99] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 1–25. Springer-Verlag, 1998.

[100] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *LNCS*, pages 18–29, Berlin, 2000. Springer-Verlag.

[101] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. BETA Working Paper Series, WP 47. Technical report, Eindhoven University of Technology, Eindhoven, The Nederlands, 2000.

[102] W3C, http://www.w3.org/TR/html/. *Hyper Text Markup Language (HTML), W3C Recommendation*.

[103] L. Wagenhals. *Course of Action Development and Evaluation using Discrete Event System Models of Influence Nets*. PhD Dissertation, GMU/C3I/SAL-212-TH. C3I Center, George Mason University, Fairfax, VA, USA, January 2000.

[104] L. Wagenhals, S. Haider, and A. Levis. Synthesizing Executable Models of Object Oriented Architectures. In C. Lakos, R. Esser, L. Kristensen, and J. Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 85–94. Australian Computer Society Inc., June 2002.

[105] L. Wagenhals and A. Levis. Course of Action Development and Evaluation. In *Proceedings of Command and Control Research and Technology Symposium*, Naval Academy, Annapolis, MD, USA, 2000.

[106] L. Wagenhals and A. Levis. Modeling Effects-Based Operations in Support of War Games. In A. Sisti and D. Trevisani, editors, *Enabling Technology for Simulation Science V*, volume 4367, pages 365–376. International Society for Optical Engineering, September 2001.

[107] L. Wagenhals, I. Shin, and A. Levis. Effects-based Course of Action Analysis in Support of War Games. In A. Sisti and D. Trevisani, editors, *Enabling Technology for Simulation Science VI*, volume 4716. International Society for Optical Engineering, April 2002.

[108] L. Wagenhals, I. Shin, and A. Lewis. Creating Executable Models of Influence Nets with Colored Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):168–181, December 1998.

[109] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl, 2nd edition.* O'Reilly & Associates, Inc., September 1996.

[110] WOSIT. Online: http://www.mitre.org/technology/wosit/wosit/.

[111] L. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[112] L. Zhang, L. Kristensen, C. Janczura, G. Gallasch, and J. Billington. A Coloured Petri Net based Tool for Course of Action Development and Analysis. In C. Lakos, R. Esser, L. Kristensen, and J. Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 125–134. Australian Computer Society Inc., June 2002.