

Chapter 1

Introduction to Modelling and Validation

System development and engineering is a comprehensive discipline involving a multitude of activities such as requirements engineering, design and specification, implementation, testing, and deployment. An increasing number of system development projects are concerned with concurrent systems. There are numerous examples of this, ranging from large-scale systems, in the areas of telecommunication and applications based on Internet technology, to medium- or small- scale systems, in the area of embedded systems.

Section 1.1 introduces the basic ideas and motivation for modelling in system development, and Sect. 1.2 gives a high-level overview of the CPN modelling language. Section 1.3 discusses the role of abstraction and visualisation when one is constructing models of concurrent systems. Section 1.4 presents the benefits of formal modelling languages and verification. Section 1.5 gives an overview of the main features of CPN Tools. Finally, Sect. 1.6 provides an overview of the results from four industrial projects using CP-nets. A more detailed description of the projects will be given in Chap. 14.

1.1 Modelling and System Development

The development of concurrent systems is particularly challenging. A major reason is that these systems possess concurrency and non-determinism which means that the execution of such systems may proceed in many different ways, for example, depending on whether messages are lost during transmission, the scheduling of processes, and the time at which input is received from the environment. Hence, such systems have an astronomical number of possible executions. It is extremely easy for a human designer to miss some important interaction patterns when designing such a system, leading to gaps or malfunctions in the system design. As a result, concurrent systems are, by nature, complex and difficult to design, test, and debug. Furthermore, for many concurrent systems such as those integrated into nuclear power plants, aircraft control systems, and hospital life support equipment, it is essential that the system works correctly from the very beginning. To cope with the

complexity of modern concurrent systems, it is therefore crucial to provide methods that enable the debugging and testing of central parts of system designs prior to implementation and deployment.

One way to approach the challenge of developing concurrent systems is to build a *model* of the system. Modelling is a universal technique that can be used across many of the activities in system development. Many modelling languages have been suggested, and many are being used for system development. One prominent example is the Unified Modeling Language (UML) [94] which is becoming the de-facto standard modelling language of the software industry and which supports modelling of the structure and behaviour of systems. The focus of this textbook is on executable models that can be used to *simulate* the behaviour of systems.

The act of constructing a model of a system to be developed is typically done in the early phases of system development, and is also known from other disciplines, such as when engineers construct bridges and architects design buildings. For example, architects make architectural drawings and may build three-dimensional models in cardboard, plastic, or plywood, or use computerised 3D animation to visualise a building. The purpose of these different models is to get a better impression of the building. This allows the architect and the intended owners and users of the building to imagine what the building will look like and how it will function, for example, whether some corridors are too narrow or some doors so close to each other that they may create dangerous situations. The main motivation behind such models is that it is obviously preferable to correct design errors and other shortcomings before the construction of the real building commences.

When a new concurrent system is being designed or an existing one is being investigated, there are similar reasons why it is beneficial to build a model of it and to build it as early as possible.

- *Insight.* The act of constructing the model and simulating it usually leads to significant new insights into the design and operation of the system considered. Typically the modeller gains a more elaborate and complete understanding of the system than what can be obtained by conventional means, for example, by reading design documents. The same applies to people to whom a model of a system is being presented. This insight often results in a simpler and more streamlined design. By investigating a model, similarities can be identified that can be exploited to unify and generalise the design and make it more logical, or we may get ideas to improve the usability of the system.
- *Completeness.* The construction of an executable model usually leads to a more complete specification of the design. Gaps in the specification of the system will become explicit as they will prohibit the model from being executed because certain parts are missing, or when the model is simulated, the designers and users will find that certain expected events are not possible in the current state. Modelling also leads to a more complete identification and understanding of the requirements to be placed on the system, in particular because models can be used to mediate discussions among designers and users of the system.
- *Correctness.* When simulations of a model are made a number of design errors and flaws are usually detected. Since the modeller is able to control the execution

of a model, unlike the real system, problematic scenarios can be reproduced, and it can be checked whether a proposed modification of the design does indeed fix an identified error or improves the design in the way intended. Checking a number of different scenarios by means of simulations does not necessarily lead to correct designs – there may simply be too many scenarios to investigate or the modeller may fail to notice the existence of some important scenarios. However, a systematic investigation of scenarios often significantly decreases the number of design errors.

The construction of a model of a system design typically means that more effort is spent in the early phases of system development, i.e., in requirements engineering, design, and specification. This additional investment is in most cases justified by the additional insight into the properties of the system which can be gained prior to implementation. Furthermore, many design problems and errors can be discovered and resolved in the requirements and design phase rather than in the implementation phase. Finally, models are in most cases simpler and more complete than traditional design documents, which means that the construction and exploration of the model can result in a more solid foundation for doing the implementation. This may in turn shorten the implementation and test phases significantly and decrease the number of flaws in the final system.

1.2 Coloured Petri Nets

Coloured Petri Nets (CP-nets or CPNs) [60, 61, 63] is a graphical language for constructing models of concurrent systems and analysing their properties. CP-nets is a discrete-event modelling language combining the capabilities of Petri nets [88, 93] with the capabilities of a high-level programming language. Petri nets provide the foundation of the graphical notation and the basic primitives for modelling concurrency, communication, and synchronisation. The CPN ML programming language, which is based on the functional programming language Standard ML [84, 102], provides the primitives for the definition of data types, for describing data manipulation, and for creating compact and parameterisable models. The CPN modelling language is a general-purpose modelling language, i.e., it is not aimed at modelling a specific class of systems, but is aimed towards a very broad class of systems that can be characterised as concurrent systems. Typical application domains of CP-nets are communication protocols, data networks, distributed algorithms, and embedded systems. CP-nets are, however, also applicable more generally for modelling systems where concurrency and communication are key characteristics. Examples of these are business processes and workflows, manufacturing systems, and agent systems. An updated list of examples of industrial applications of CP-nets within various domains is available via [40].

Petri Nets are traditionally divided into *low-level Petri Nets* and *high-level Petri Nets*. CP-nets belong to the class of high-level Petri Nets which are characterised by the combination of Petri Nets and programming languages. Low-level Petri Nets

(such as Place/Transition Nets [30]) are primarily suited as a theoretical model for concurrency, although certain classes of low-level Petri Nets are often applied for modelling and verification of hardware systems [111]. High-level Petri Nets (such as CP-nets and Predicate/Transition Nets [45]) are aimed at practical use, in particular because they allow the construction of compact and parameterised models. High-level Petri Nets is an ISO/IEC standard [7], and the CPN modelling language and supporting computer tools conform to this standard.

CPN models are executable and are used to model and specify the behaviour of concurrent systems. A CPN model of a system is both state and action oriented. It describes the states of the system and the events (transitions) that can cause the system to change state. By performing simulations of the CPN model, it is possible to investigate different scenarios and explore the behaviour of the system. Very often, the goal of performing simulations is to debug and investigate the system design. CP-nets can be simulated interactively or automatically. An interactive simulation is similar to single-step debugging. It provides a way to ‘walk through’ a CPN model, investigating different scenarios in detail and checking whether the model works as expected. During an interactive simulation, the modeller is in charge and determines the next step by selecting between the enabled events in the current state. It is possible to observe the effects of the individual steps directly in the graphical representation of the CPN model. This is similar to an architect deciding the exact route to follow while performing an interactive walk through a 3D computer model of a building. Automatic simulation is similar to program execution and the purpose is to execute the CPN model as fast and efficiently as possible, without detailed human interaction and inspection. Automatic simulation is typically used for testing and performance analysis. For testing purposes, the modeller typically sets up appropriate breakpoints and stop criteria. For performance analysis the model is instrumented with data collectors to collect data on the performance of the system.

Time plays a significant role in a wide range of concurrent systems. The correct functioning of some systems depends crucially on the time taken by certain activities, and different design decisions may have a significant impact on the performance of a system. CP-nets include a concept of time that makes it possible to capture the time taken by events in the system. This time concept also means that CP-nets can be applied to simulation-based performance analysis, where performance measures such as delays, throughput, and queue lengths in the system are investigated, and for modelling and validation of real-time systems.

The development of CP-nets has been driven by the desire to develop an industrial strength modelling language – theoretically well founded and at the same time versatile enough to be used in practice for systems of the size and complexity found in typical industrial projects. CP-nets, however, is not a modelling language designed to replace other modelling languages (such as UML). In our view, CP-nets should be used as a supplement to existing modelling languages and methodologies and can be used together with these or even integrated into them.

CP-nets is one of many modelling languages [14] developed for concurrent and distributed systems. Other prominent examples are Statecharts [50] as supported by, for example, the VisualState tool [103], the Calculus of Communicating Systems

[83] as supported by, for example, the Edinburgh Concurrency Workbench [32], Timed Automata [1] as supported by, for example, the UPPAAL tool [76], Communicating Sequential Processes [52] as supported by, for example, the FDR tool [41], and Promela [54], as supported by the SPIN tool [96].

CP-nets has been under development by the CPN group at Aarhus University, Denmark since 1979. The first version was part of the PhD thesis of Kurt Jensen and was presented in [59]. It was inspired by the pioneering work of Hartmann Genrich and Kurt Lautenbach on Predicate/Transition Nets [46]. Since then, the CPN group has been working on the consolidation of the basic model, extensions to cope with modules and time, and methods for analysis by means of state spaces and simulation-based performance analysis. Simultaneously the group has developed and distributed industrial-strength computer tools, such as Design/CPN [28] and CPN Tools [25], and we have conducted numerous application projects [40] where CP-nets and their tools have been used together with industrial partners. For a more detailed description of the origin of CP-nets and their relation to other kinds of Petri Nets, the reader is referred to the bibliographical remarks in Chap. 1 of [60]. Numerous people have contributed to the development of CP-nets and their tools. This includes the many people who have worked in the CPN group and the hundreds of tool users who have proposed valuable extensions and improvements.

1.3 Abstraction and Visualisation

When a model is constructed, abstractions are made, which means that a number of details are omitted. As an example, it is unlikely that an architect constructing an architectural model of a building using cardboard, plastic, or plywood, will include any information about the plumbing and wiring of the building. These things are irrelevant for the purpose of this kind of model, which usually is to be able to judge the aesthetics of the architectural design. However, the architect will construct other models which contain a detailed specification of the wiring and plumbing. When constructing a model, the first questions to ask ourselves should be: What is the purpose? What do we want to learn about the system by making this kind of model? What kinds of properties are we interested in investigating? Without initially answering these questions in some detail, it is impossible to make a good model, and we shall be unable to decide what should be included in the model, and what can be abstracted away without compromising the correctness of the conclusions that will be drawn from investigating the model. Finding the appropriate abstraction level at different points in the development of systems is one of the arts of modelling.

The CPN language has few, but powerful modelling primitives, which means that relatively few constructs must be mastered to be able to construct models. The modelling primitives also make it possible to model systems and concepts at different levels of abstraction. CPN models can be structured into a set of modules. This is particularly important when one is dealing with CPN models of large systems. The modules interact with each other through a set of well-defined interfaces, in a way

similar to that of programming languages. The concept of modules in CP-nets is based on a hierarchical structuring mechanism, which allows a module to have sub-modules, allows a set of modules to be composed to form a new module, and allows reuse of submodules in different parts of the model. This enables the modeller to work both top-down and bottom-up when constructing CPN models. By means of the structuring mechanism, it is possible to capture different abstraction levels of the modelled system in the same CPN model. A CPN model which represents a high level of abstraction is typically constructed in the early stages of design or analysis. This model is then gradually refined to yield a more detailed and precise description of the system under consideration. The fact that it is possible to abstract away from many implementation details and gradually refine the system design implies that constructing a CPN model can be a very cost-effective way of obtaining a first executable prototype of a system.

Visualisation is a technique which is closely related to simulation of CPN models. An important application of visualisation is that it allows the presentation of design ideas and analysis results using concepts from the application domain. This is particularly important in discussions with people unfamiliar with CP-nets. The CPN modelling language includes several means for adding application domain graphics on top of the CPN model. This can be used to abstractly visualise the execution of the CPN model in the context of the application domain. One example is the use message sequence charts (or time sequence diagrams) [15] to visualise the exchange of messages in the execution of a communication protocol. Furthermore, observing every single step in a simulation is often too detailed a level of observation of the behaviour of a system. It provides the observer with an overwhelming amount of detail, particularly for large CPN models. By means of visual feedback from simulations, information about the execution of the system can be obtained at a more adequate level of detail.

1.4 Formal Modelling and Verification

CPN models are formal, in the sense that the CPN modelling language has a mathematical definition of both its syntax and its semantics. Such models can be manipulated by a computer tool and can be used to *verify* system properties, i.e., prove that certain desired properties are fulfilled or that certain undesired properties are guaranteed to be avoided. The formal representation is the foundation for the definition of the various behavioural properties and the analysis methods. Without the mathematical representation it would have been impossible to develop a sound and powerful CPN language.

Formal verification is, by its nature, different from and supplements the kind of informal analysis performed when individual scenarios are inspected by means of simulation. Verification involves a mathematical formulation of a property and a computer-assisted proof that this property is fulfilled by the model. When verifying system properties, it is also necessary to argue that the model captures those aspects

that are relevant for the property we are verifying. It must be ensured that the verified properties are really those that we want the system to possess. This means that formal verification is always accompanied by informal justifications.

Verification of CPN models and system properties is supported by the *state space method*. The basic idea underlying state spaces is to compute all reachable states and state changes of the CPN model and represent these as a directed graph, where nodes represent states and arcs represent occurring events. State spaces can be constructed fully automatically. From a constructed state space, it is possible to answer a large set of verification questions concerning the behaviour of the system, such as absence of deadlocks, the possibility of always being able to reach a given state, and the guaranteed delivery of a given service.

One of the main advantages of state spaces is that they can provide counterexamples (or error traces) giving detailed debugging information specifying why an expected property does not hold. Furthermore, state spaces are relatively easy to use, and they have a high degree of automation. The ease of use is primarily due to the fact that it is possible to hide a large portion of the underlying complex mathematics from the user. This means that, quite often, the user is required only to formulate the property which is to be verified and then apply a computer tool. The main disadvantage of using state spaces is the *state explosion problem* [106]: even relatively small systems may have an astronomical or even infinite number of reachable states, and this is a serious problem for the use of state spaces in the verification of real-life systems. A wide range of state space reduction methods have therefore been developed for alleviating the state explosion problem inherent in state space-based verification. It is also possible to use state spaces in conjunction with other analysis methods for CP-nets, such as place invariants and net structure reductions.

The capability of working at different levels of abstraction is one of the keys to making formal analysis of CP-nets possible. By abstraction it is possible to make very large, detailed models tractable for state space analysis. The state space method of CP-nets can also be applied to timed CP-nets. Hence, it is also possible to verify the functional correctness of systems modelled by means of timed CP-nets.

The formal definition of CP-nets implies that CPN models are unambiguous and hence provides a precise specification of the design. This is in contrast to design specifications written in natural language, which are inherently ambiguous. Having precise, unambiguous specifications is generally desirable, and it is crucial in many areas such as the development of open protocol standards, where precise specifications are required to ensure interoperability between implementations made by different vendors.

It should be stressed that for the practical use of CP-nets and their supporting computer tools, it suffices to have an intuitive understanding of the syntax and semantics of the CPN modelling language. This is analogous to the situation for ordinary programming languages that are successfully applied by programmers who are usually not familiar with the formal, mathematical definitions of the languages.

The practical application of CP-nets typically relies on a combination of interactive- and automatic simulation, visualisation, state space analysis, and performance analysis. These activities in conjunction result in a *validation* of the system

under consideration, in the sense that it is possible to justify the assertion that the system has the desired properties, and a high degree of confidence in and understanding of the system has been obtained.

1.5 CPN Tools

The practical application of modelling and validation relies heavily on the existence of computer tools supporting the construction and manipulation of models.

CPN Tools is a tool for the editing, simulation, state space analysis, and performance analysis of CPN models. CPN Tools supports untimed and timed hierarchical CPN models. CPN Tools is used by more than 8,000 users in 140 different countries and is available for Windows XP, Windows Vista, and Linux. A licence for CPN Tools can be obtained free of charge via the CPN Tools Web pages [25]. Below, we provide a very brief introduction to CPN Tools. The CPN Tools Web pages contain an elaborate set of manuals on how to use the tool.

The user of CPN Tools works directly with the graphical representation of the CPN model. The graphical user interface (GUI) of CPN Tools has no conventional menu bars and pull-down menus, but is based on interaction techniques, such as *tool palettes* and *marking menus*. Figure 1.1 provides a screenshot of CPN Tools. The rectangular area to the left is an *index*. It includes the Tool box, which is available for the user to manipulate the declarations and modules that constitute the CPN model. The Tool box includes tools for creating, copying, and cloning the basic elements of CP-nets. It also contains a wide selection of tools to manipulate the graphical layout and the appearance of the objects in the CPN model. The latter set of tools is very important in order to be able to create readable and graphically appealing CPN models. The remaining part of the screen is the *workspace*, which in this case contains four *binders* (the rectangular windows) and a circular pop-up menu.

Each binder holds a number of items which can be accessed by clicking the tabs at the top of the binder (only one item is visible at a time). There are two kinds of binders. One kind contains the elements of the CPN model, i.e., the modules and declarations. The other kind contains the tools which the user applies to construct and manipulate CPN models. The tools in a tool palette can be picked up with the mouse cursor and applied. In the example shown, one binder contains three modules named Protocol, Sender, and Receiver, while another binder contains a single module, named Network, together with the declaration of the colour set NOxDATA. The two remaining binders contain four different tool palettes to Create elements, change their Style, perform Simulations, and construct State spaces.

Items can be dragged from the index to the binders, and from one binder to another binder of the same kind. It is possible to position the same item in two different binders, for example, to view a module using two different zoom factors. A circular marking menu has been popped up on top of the bottom left binder. Marking menus are contextual menus that make it possible to select among the

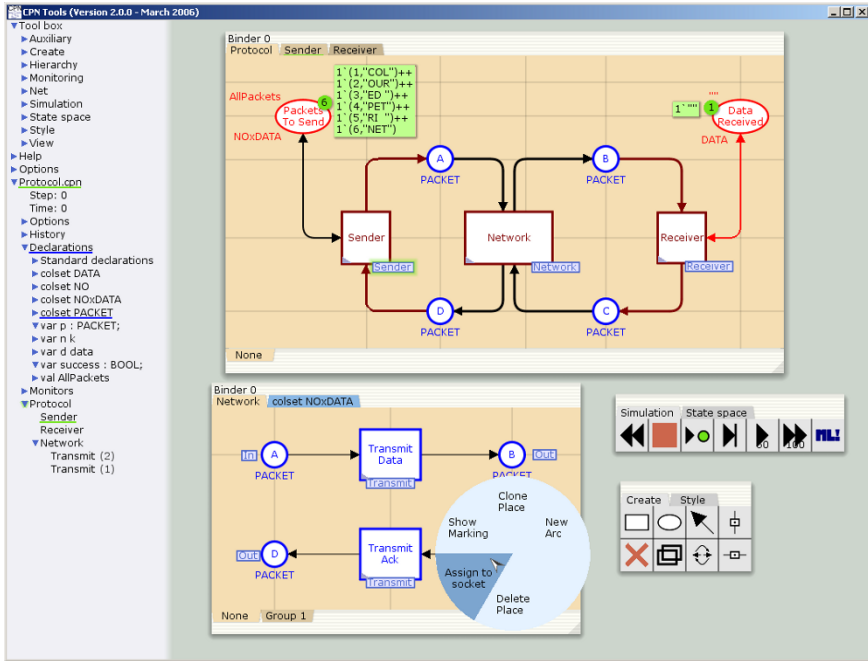


Fig. 1.1 Screenshot from CPN Tools

operations possible on a given object. In the case of Fig. 1.1, the marking menu gives the operations that can be performed on a port place object.

CPN Tools performs syntax and type checking, and error messages are provided to the user in a contextual manner next to the object causing the error. The syntax check and code generation are incremental and are performed in parallel with editing. This means that it is possible to execute parts of a CPN model even if the model is not complete, and that when parts of a CPN model are modified, a syntax check and code generation are performed only on the elements that depend on the parts that were modified. The main outcome of the code generation step is the *simulation code*. The simulation code contains the functions for inferring the set of enabled events in a given state of the CPN model, and for computing the state resulting from the occurrence (execution) of an enabled event in a given state.

CPN Tools supports two types of simulation: interactive and automatic. In an interactive simulation, the user is in complete control and determines the individual steps in the simulation, by selecting between the enabled events in the current state. CPN Tools shows the effect of executing a selected step in the graphical representation of the CPN model. In an automatic simulation the user specifies the number of steps that are to be executed and/or sets a number of stop criteria and breakpoints. The simulator then automatically executes the model without user interaction by making random choices between the enabled events in the states encountered. Only the resulting state is shown in the GUI. A *simulation report* can be saved, contain-

ing a specification of the steps that occurred during an automatic simulation. The simulator of CPN Tools exploits a number of advanced data structures for efficient simulation of large hierarchical CPN models. The simulator exploits the locality property of Petri nets to ensure that the number of steps executed per second in a simulation is independent of the size of the CPN model. This guarantees that simulation scales to large CPN models.

Full state spaces, which are state spaces in their most basic form, and a collection of advanced state space methods are supported by CPN Tools. The advanced methods make it possible to alleviate the impact of the state explosion problem, which is particularly evident when state space analysis of large CPN models is conducted. CPN Tools provides several means for analysing the properties of the system under consideration using state spaces. The first step is usually to create a *state space report* containing answers to a set of standard behavioural properties of CPN models, such as the absence or presence of deadlocks and the minimum and maximum number of tokens on the individual places. In the early stages of system development, design errors are very often evident in the state space report, which can be generated fully automatically. It is also possible for the user to interactively draw selected parts of a state space and inspect the individual states and events. This can be a very effective way of debugging a system. CPN Tools implements a set of query functions that makes it possible for the user to traverse the state space in a number of ways and thereby investigate system-dependent properties. Verification of system properties based on formulating properties in temporal logic [37] and conducting model checking [21, 22] is also supported.

Simulation-based performance analysis is supported via automatic simulation combined with elaborate data collection. The basic idea of simulation-based performance analysis is to conduct a number of lengthy simulations of the model during which data about the performance of the system is collected. The data typically provides information such as the sizes of queues, the delays of packets, and the load on various components. The collection of data is based on the concept of *monitors* that allow the user to specify when data is to be collected during the individual steps of a series of automatic simulations, and what data is to be collected. The data can be written into log files for postprocessing, for example, in a spreadsheet, or a *performance report* can be saved, summarising key figures for the collected data such as averages, standard deviations, and confidence intervals. Simulation-based performance analysis typically uses *batch simulation*, which makes it possible to explore the parameter space of the model without user intervention and to conduct multiple simulations of each parameter configuration to obtain statistically reliable results.

CPN Tools includes a visualisation package [109] implemented in Java that supports the user in constructing application domain graphics on top of CPN models. Such graphics can provide an abstract application-specific presentation of the dynamics of the modelled system. They can be used to make the underlying formal CPN model fully transparent to the observer. The animation package supports several standard diagram and chart types, such as message sequence charts. The animation package also allows the user to implement additional diagram types using existing libraries.

CPN Tools also includes a collection of libraries for various purposes. One example is Comms/CPN [42], for TCP/IP communication between CPN models and external applications. CPN Tools generally has an open architecture that allows the user to extend its functionality, such as for experimenting with new state space methods. Hence, in addition to being a tool for modelling and validation, it also provides a prototyping environment for researchers interested in experimenting with new analysis algorithms.

This book relies on a series of relatively simple CPN models of a communication protocol, gradually enhanced and modified using the concepts and tools presented in the following chapters. An industrial case study in Sect. 14.1 shows how much more complex protocols can be modelled and validated.

1.6 Industrial Applications

An overview of industrial applications of CP-nets can be obtained via the Web pages [40] which contain references to more than 100 published papers on CPN projects. In Chap. 14, we will present four representative projects where CP-nets and their supporting computer tools have been used for system development in an industrial context. The projects have been selected to illustrate the fact that CP-nets can be used in many different phases of system development, ranging from requirements specification to design, validation, and implementation. The CPN models presented were constructed in joint projects between our research group at Aarhus University and industrial partners. Chapter 14 provides a detailed description of the four projects. Below, we shall only give an overview of the most important results.

The first project was concerned with the development of the Edge Router Discovery Protocol (ERDP) at Ericsson Telebit. In the project, a CPN model was constructed that constituted a formal executable specification of ERDP. Simulation and message sequence charts were used in initial investigations of the protocol's behaviour. Then state space analysis was applied to conduct a formal verification of the key properties of ERDP.

The application of CP-nets in the development of ERDP was successful for three main reasons. Firstly, it was demonstrated that the CPN modelling language and supporting computer tools are powerful enough to specify and analyse a real-world communication protocol and that they can be integrated into a conventional protocol development process. Secondly, the modelling, simulation, and subsequent state space analysis all helped to identify several omissions and errors in the design, demonstrating the benefits of using formal techniques in a protocol design process. Finally, the effort of constructing the CPN model and conducting the state space analysis was represented by approximately 100 person-hours. This is a relatively small investment compared with the many issues that were identified and resolved early as a consequence of constructing and analysing the CPN model.

The second project was concerned with specifying the business processes at Aarhus County Hospital and their support by a new IT System, called the Pervasive

Health Care System (PHCS). A CPN model of PHCS was used to engineer requirements for the system. Behavioural visualisation driven by a CPN model was used to visualise system behaviour and enable the engineering of requirements through discussions with people who were not familiar with the CPN modelling language.

The project demonstrated that CPN models are able to support various requirements engineering activities. One of the main motivations for the approach chosen for PHCS was to build on top of prose descriptions of work processes and the proposed computer support, consolidated as UML use cases. The stakeholders of PHCS were already familiar with the UML use cases from earlier projects. The visualisations enabled users such as nurses and doctors to be actively engaged in specification analysis and elicitation, which is crucial. User participation increases the probability that a system is ultimately built that fits with the future users' work processes.

The third project was concerned with the design and analysis of the BeoLink system at Bang & Olufsen. A timed CPN model was developed, specifying the lock management subsystem which is responsible for the basic synchronisation of the devices in the BeoLink system. State spaces were used to verify the lock management system.

The project demonstrated the use of CP-nets for modelling and validating a real-time system, i.e., a system where the correctness of the system depends on timing information. Engineers at Bang & Olufsen were given a four-day course on CP-nets, enabling them to construct large parts of the CPN model. This demonstrates (as also seen in other projects) that a relatively short introduction is required to get started on using CP-nets in industrial projects. In the original BeoLink project, only the initialisation phase of the lock management protocol was verified using state spaces. Since then, a number of advanced state space methods have been developed and implemented, and these methods have been used to verify configurations of the BeoLink system that could not be verified using ordinary state spaces. It has also been demonstrated that the advanced state space methods can be used simultaneously to get a better reduction than obtainable from either method in isolation.

The fourth project was concerned with the development of a military scheduling tool (COAST) in which the analysis capabilities are based on state space methods. CPN modelling was used to conceptualise and formalise the planning domain to be supported by the tool. Later on, the CPN model was extracted in executable form from CPN Tools and embedded directly into the server of COAST together with a number of tailored state space analysis algorithms.

The project demonstrated how a CPN model can be used for the implementation of a computer tool thereby overcoming the usual gap between the design and the final implementation. It also demonstrated the value of having a full programming-language environment in the form of the Standard ML compiler integrated into CPN Tools. This allowed a highly compact and parameterisable CPN model to be constructed, and allowed the CPN model to become the implementation of the COAST server. It also made it possible to extend the COAST server with the specialised algorithms required to extract task schedules from the generated state spaces.