Chapter 3

Behavioural Visualisation of Formal Models

Until now, we have considered the model in Fig. 1.5 as an example of a coloured Petri net. That is not really true. In fact, according to the ISO standard for Petri nets [87], a coloured Petri net (in the standard called a high-level Petri net) is a septuple of types, places, transitions, a type function assigning types to places and transitions, backward and forward incidence functions indicating how many tokens are consumed respectively produced when each transition is executed, and an initial marking. Using this definition and renaming places to P_1–P_6 and transitions to T_1–T_4 to keep the figure more compact, the network protocol in Fig. 1.5 would look like the tuple in Fig. 3.1. In fact the tuple in Fig. 3.1 only shows the system in the initial state. To also show the dynamic behaviour of the protocol, we would need to give a mapping like the initial marking (the last element of the tuple) for each state encountered. Obviously the description in Fig. 1.5 is much more readable than the one in Fig. 3.1, which is exactly why the graphical notation in Fig. 1.5 was invented and is being used in practise.

The formal description is important when reasoning about the formalism, e.g., when proving that some extension is just syntactic as is, e.g., the case for Christensen and Hansen's synchronous channels [22]. When we want to create the model or explore it using simulation, we do not need the formal definition explicitly, and will prefer the graphical notation instead as it makes the behaviour of the system much clearer. In Fig. 3.2 we see three different layers of formal models; the *mathematical model* in Fig. 3.1 is located in the bottom layer and is used by developers of the formalism. This layer is used to reason about the formalism and to develop analysis methods which works on all concrete models. The layer above it, the middle layer, is represented by the *graphical model* in Fig. 1.5. It consists of concrete models, and is primarily used by the formal methods expert, who focuses on creating formal models. The graphical model shows all places as ellipses and all transitions as rectangles. The type of places is shown next to the place as is the initial marking. The backward and forward incidence functions are shown as arcs from places to transitions (in the case of the backwards incidence function) and arcs from transitions to places (for the forward incidence function) and the right-hand side of the lambda-expression corresponding to the arc is shown near the arc if it is not the empty multi-set. The top layer, the visualisation, is used primarily by a domain expert to validate that the formal model corresponds to the intended system. This is the job of a domain expert, as he has extensive knowledge of the domain of the model, while the formal methods expert will seldom

$$
\begin{array}{ll}
(& \{ \quad ID, PACKET, PACKETS, PACKETxPACKETS \quad \}, \\
& \{ \quad P_1, P_2, P_3, P_4, P_5, P_6 \quad \}, \\
& \{ \quad T_1, T_2, T_3, T_4 \quad \}, \\
& \{ \quad P_1 \mapsto PACKET, P_2 \mapsto ID, P_3 \mapsto PACKET, \\
& \quad P_4 \mapsto PACKET, P_5 \mapsto ID, P_6 \mapsto PACKETS, \\
& \quad T_1 \mapsto PACKET, T_2 \mapsto PACKET, T_3 \mapsto PACKET, \\
& \quad T_4 \mapsto PACKETxPACKETS \quad \}, \\
& \{ \quad (P_1, T_1) \mapsto \lambda x.1`x, (P_1, T_2) \mapsto \lambda x.\emptyset, (P_1, T_3) \mapsto \lambda x.\emptyset, \\
& \quad (P_1, T_4) \mapsto \lambda x.\emptyset, (P_2, T_1) \mapsto \lambda(x, y).1`x, (P_1, T_2) \mapsto \lambda x.\emptyset, \\
& \quad (P_2, T_3) \mapsto \lambda x.\emptyset, (P_2, T_4) \mapsto \lambda(x, y).1`x, (P_3, T_1) \mapsto \lambda x.\emptyset, \\
& \quad (P_3, T_2) \mapsto \lambda x.1`x, (P_3, T_3) \mapsto \lambda(x, y).1`x, (P_3, T_4) \mapsto \lambda x.\emptyset, \\
& \quad (P_4, T_1) \mapsto \lambda x.\emptyset, (P_4, T_2) \mapsto \lambda x.\emptyset, (P_4, T_3) \mapsto \lambda x.\emptyset, \\
& \quad (P_4, T_4) \mapsto \lambda x.1`x, (P_5, T_1) \mapsto \lambda x.\emptyset, (P_5, T_2) \mapsto \lambda x.\emptyset, \\
& \quad (P_5, T_3) \mapsto \lambda((x, y), z).1`x, (P_5, T_4) \mapsto \lambda x.\emptyset, (P_6, T_1) \mapsto \lambda x.\emptyset, \\
& \quad (P_6, T_2) \mapsto \lambda x.\emptyset, (P_6, T_3) \mapsto \lambda(x, y).1`y, (P_6, T_4) \mapsto \lambda x.\emptyset \quad \}, \\
& \{ \quad (P_1, T_1) \mapsto \lambda x.1`x, (P_1, T_2) \mapsto \lambda x.\emptyset, (P_1, T_3) \mapsto \lambda x.\emptyset\}, \\
& \quad (P_1, T_4) \mapsto \lambda x.\emptyset, (P_2, T_1) \mapsto \lambda(x, y).1`x, (P_1, T_2) \mapsto \lambda x.\emptyset, \\
& \quad (P_2, T_3) \mapsto \lambda x.\emptyset, (P_2, T_4) \mapsto \lambda(x, y).1`(x + 1), (P_3, T_1) \mapsto \lambda x.1`x, \\
& \quad (P_3, T_2) \mapsto \lambda x.\emptyset, (P_3, T_3) \mapsto \lambda x.\emptyset, (P_3, T_4) \mapsto \lambda x.\emptyset, \\
& \quad (P_4, T_1) \mapsto \lambda x.\emptyset, (P_4, T_2) \mapsto \lambda x.\emptyset, (P_4, T_3) \mapsto \lambda((x, y), z).1`(x, \text{""}), \\
& \quad (P_4, T_4) \mapsto \lambda x.\emptyset, (P_5, T_1) \mapsto \lambda x.\emptyset, (P_5, T_2) \mapsto \lambda x.\emptyset \\
& \quad (P_5, T_3) \mapsto \lambda((x, y), z).1`(x + 1), (P_5, T_4) \mapsto \lambda x.\emptyset, (P_6, T_1) \mapsto \lambda x.\emptyset, \\
& \quad (P_6, T_2) \mapsto \lambda x.\emptyset, (P_6, T_3) \mapsto \lambda(x, y).y \wedge \wedge x, (P_6, T_4) \mapsto \lambda x.\emptyset \quad \}, \\
& \{ \quad P_1 \mapsto 1`(1, \text{"Formal"}) + +1`(2, \text{" model"}), P_2 \mapsto 1`1, \\
& \quad P_3 \mapsto \emptyset, P_4 \mapsto \emptyset, P_5 \mapsto 1`1, P_6 \mapsto 1`[] \quad \} \quad)
\end{array}
$$

Figure 3.1: The network protocol from Fig. 1.5 as it looks using the formal definition of [87].

know enough about the domain to validate all details of the model. While the graphical representation in Fig. 1.5 is easier to read for coloured Petri nets experts, it is not that intuitive for other people, and it may not be evident to a network engineer that the model in Fig. 1.5 actually is a network protocol. The problem only gets worse if the model is larger or the domain expert knows even less about formal models, for example if the domain expert is a nurse. We can then use the method in Fig. 1.3 to construct a graphical model from the specification and let the domain expert validate that the formal model corresponds to the specification using the visualisation.

Relating the Model-View-Controller (MVC) [100] design pattern [54] from Fig. 1.10 to the 3 layers of use of formal models shown in Fig. 3.2, we can think of the lowest level, the mathematical model, as the model (in MVC terms) of the system. We can think of the graphical model as the view. A tool implementing simulation of a formal model will have an internal representation corresponding to the formal definition, as this is required to implement the correct semantics of the modelling language. It may have a graphical user interface which

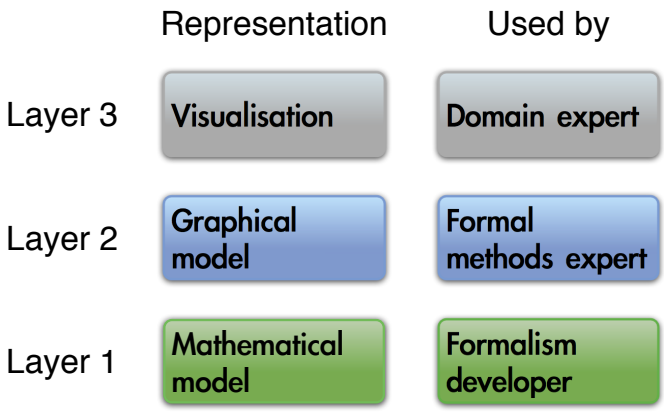


Figure 3.2: Three layers of use of formal models.

allows the user to manipulate the formal model using the graphical model layer of Fig. 3.2. The controller allows the modeller to modify the model (thereby incrementally building the desired model) and to simulate the model. The basic idea of most visualisation tools is that we add a new view on the model. The view will often be more simplistic than the graphical representation of the underlying formal model, the graphical model. Some tools will even allow the user to manipulate the execution of the model.

The rest of this chapter is structured as follows: In Sect. 3.1 we will give a brief survey of different visualisation tools aimed at various formalisms. Sections 3.2 to 3.4 summarise papers co-authored by the author of this thesis. Section 3.2 summarises our paper [T3], which describes the BRITNeY Suite, a formalism-independent user-extensible platform and tool for creating visualisations of formal models. Section 3.3 summarises our paper [T4], which provides an industrial case-study where a visualisation of a formal model has been developed and the BRITNeY Suite visualisation tool put into practical use. Section 3.4 summarises our paper [T5], which provides a formalism-independent abstract framework for visualisations based on game-theory. This framework gives a formal definition of a visualisation of a formal model lifting visualisations above an ad-hoc expert-level to a more precise, easier accessible level. Finally, in Sect. 3.5, we sum up the contribution of our papers [T3], [T4], and [T5] and provide directions for future work.

3.1 Approaches to Visualisation

Several tools supporting the methodology in Fig. 1.3 exist. In this section we will describe some of them and discuss strengths and weaknesses of each.

TU Eindhoven's ExSpect [50] is a tool for modelling based on coloured Petri nets. ExSpect allows the user to view the state of models by associating widgets with places of the model, and allows users to asynchronously interact with the model using simple widgets. Widgets can, e.g., show the number of tokens available on a certain place or add new tokens to places. This makes it possible to inspect the state of the system using well-known widgets like counters and gauges, and stimulate the execution by pushing buttons or entering text. Visualisations are created on a dashboard by dragging in the desired widgets, making it very easy to create visualisations. The disadvantage of this approach is, firstly, that it is specific to coloured Petri nets (as it relies on tokens with types)

and, secondly, that input from the user is made by switching from one state of the system to another without formally executing a transition in the model. This is problematic because the visualisation not only reflects the behaviour of the formal model, it also changes it, which makes formal verification of the underlying formal model irrelevant, as the behaviour of the formal model can be very different from the behaviour of the formal model with a visualisations. Visualisation is completely integrated in the ExSpect tool, which makes it impossible to extend it or use the visualisations with other tools or formalisms. Finally, this approach only allows users to create visualisations using a predefined set of widgets, thereby making a "cartoon-like" visualisation like the one in Fig. 1.13 impossible.

Rasmussen and Singh's MIMIC/CPN [141] is a library which facilitates visualisation of coloured Petri net models created using DESIGN/CPN [37], a tool for editing, simulating and analysing coloured Petri nets. It provides an API which can be used to define and update visualisations. By annotating a CPN model, functions of the API is called during execution of the model. Visualisations can be created using a standard drawing program, so it is easy for even inexperienced users to layout a visualisation. The disadvantage of this approach is that it is very inconvenient to have to change the model in order to add a visualisation and the changes unnecessarily clutter the model. Furthermore, MIMIC/CPN mainly focuses on state changes of the system, and everything shown to the user must be formulated as explicit updates, so it is not possible to easily monitor the value of, e.g., a counter like in ExSpect. Also, the library is very low-level, as it only allows the model to display, hide, and change the position of items previously created using the editor or using the API. The only higher-level widget supported is an ability to prompt the user for a string value, and the only other way for the user to provide input to the model is to click on buttons defined in the visualisation. Like ExSpect, MIMIC/CPN can only be used in conjunction with a single tool, namely DESIGN/CPN, but unlike ExSpect it is possible (yet very tedious) to extend the tool using Standard ML. Finally, MIMIC/CPN is unable to handle asynchronous input, which must be simulated by polling.

LTSA [116] is a tool for modelling using labelled transition systems developed by Magee and Kramer. LTSA allows users to animate models using a library called SceneBeans [117, 149] developed by Pryce and Magee. Visualisations are tied to models by associating animation activities with transition labels. Visualisations are specified using an XML file. The SceneBeans library relies on Java beans [88], which is a Java component framework, and is thus very extensible, as it is possible to extend the library with new beans. The method is nice and declarative, but it it very cumbersome to write the visualisations as XML files. Like MIMIC/CPN, the model is able to display or hide already created objects of the visualisation, and can additionally move the objects around along paths. Visualisations created using the SceneBeans library are unable to add new objects to the visualisations. The SceneBeans library can be used without LTSA, but not in conjunction with LTSA models.

Kindler and Páles' PNVis [99] is an add-on for Weber and Kinder's Petri Net Kernel [169], a modular tool for editing Petri nets. PNVis associates tokens with 3D objects and places with locations in a 3D world. The geometry of the 3D world is described using an XML file, and the look of the objects is described using VRML [86]. The visualisation is tied to the model by annotating the model with inscriptions identifying places with locations in the 3D world and tokens with objects. PNVis is suitable for modelling physical systems, but not aimed at systems that do not immediately have a physical counter-part. Furthermore the way visualisations are tied to the formal model requires, of

course, that the model is a Petri net. While it is easy to create object descriptions thanks to many available VRML editors, it is cumbersome to create the description of the world using XML files.

Harel and Marelly's Play-Engine [66] allows a prototype of a program to be implemented by inputting scenarios (play-in) via an application-specific GUI. The resulting program can then be executed (play-out). Compared to the approach of the other described tools, this makes the model implicit as it is created indirectly via the input scenarios. Furthermore, the Play-Engine relies on heavy-weight techniques to perform visualisation as the model is given implicitly. In order to decide how to execute the model, a complete model-checking step is performed in each step, which is computationally expensive.

3.2 The BRITNeY Suite Animation Tool [T3]

The BRITNeY Suite [C2, T3] was originally developed because CPN Tools [C1, 33] needed a means of creating visualisations of CPN models. As CPN Tools is written in the Beta programming language [115], it was deemed infeasible to create the visualisation tool within the tool itself due to lack of off-the-shelf libraries, meaning that all details of the tool had to be written from scratch. Instead the BRITNeY Suite was realised as an independent application written in Java. visualisation to the model. The paper [T3] is a tool presentation of the BRITNeY Suite, and this section will give the gist of the paper.

The architecture of the BRITNeY Suite, when used with CPN Tools, can be seen in Fig. 3.3. To the left we see that CPN Tools is actually composed of two components, a CPN editor and a CPN simulator. The BRITNeY Suite, to the right, consists of a main application and a number of extension plug-ins. Each extension plug-in extends the main application with new kinds of visualisation. Presently, over 20 extension plug-ins ship with the BRITNeY Suite, making it possible to create various charts, including message sequence charts (MSC) [67], gantt charts, and histograms, draw graphs in two and three dimensions, generate textual reports that can be exported to PDF files, show various dialog boxes for receiving information from and presenting information to users, and for integrating visualisations using the SceneBeans library also used by LTSA. CPN Tools communicates with the BRITNeY Suite using a standard Remote Procedure Call (RPC) [32, Sect. 5.3] mechanism called XML-RPC [170]. In order to make the communication seamless, a Stub generator component of the BRITNeY Suite injects stub code into the CPN simulator, which can then be used directly by the models.

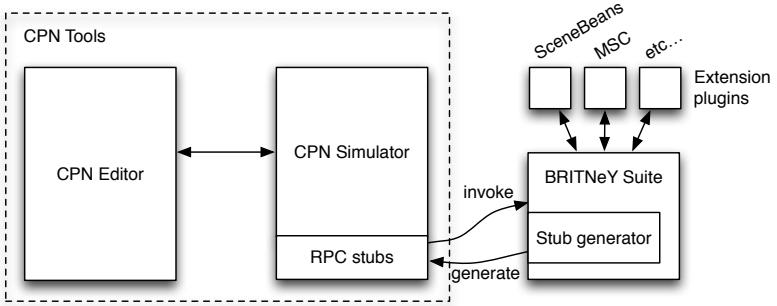
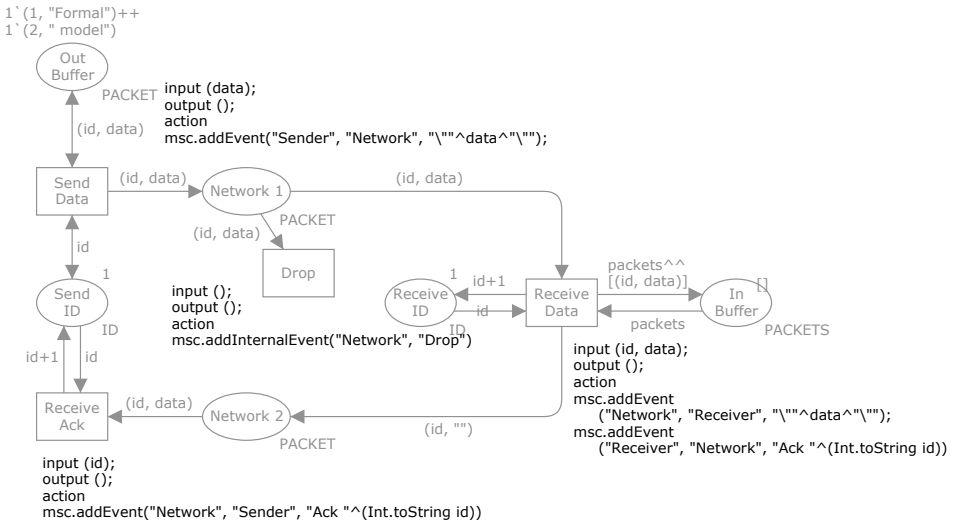


Figure 3.3: Architecture of the BRITNeY Suite when used with CPN Tools

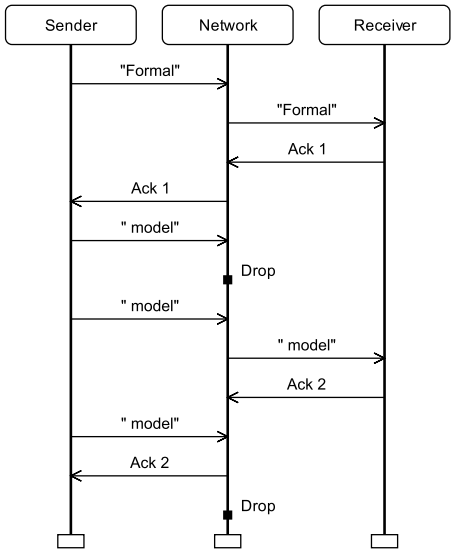
Visualisations are tied to models like in MIMIC/CPN, by annotating the model and calling functions driving the visualisation. Whenever a transition is executed, the corresponding code is executed, which may invoke a stub and consequently a function in the visualisation tool. Consider, e.g., the model in Fig. 3.4(a). The model is the same as the one in Fig. 1.5 and unchanged parts have been greyed out to highlight the changes. For each transition we have added an annotation describing how the visualisation should be updated when the transition is executed. We can, e.g., see that when the Send Data transition is executed, an event is created from Sender to Network with a label corresponding to the data sent. In Fig. 3.4(b) we see an example of the resulting visualisation. The first packet, containing "Formal", is transmitted and acknowledged successfully (this packet has sequence number 1). Then the next packet, the one containing " model", is transmitted but dropped by the network. The packet is re-transmitted and an acknowledgement is sent. Before the acknowledgement arrives at the sender, the packet is re-transmitted. The acknowledgement is then received and the network drops the outstanding packet.

While the BRITNeY Suite may seem strongly tied to CPN Tools from this description, this not the case. Any application can invoke the functions needed to drive visualisations as a standard protocol is used; the stub generator merely makes it more convenient to use the tool with CPN Tools. In fact it is possible to extend the BRITNeY Suite with a stub generator for other modelling tools as well, as the main application is not actually a solid box as indicated in Fig. 3.3, but a hierarchy of plug-ins, which can be extended. A more detailed description of the architecture of the BRITNeY Suite can be found in our workshop paper [C5]. Our paper [T3] also describes two industrial case studies where the BRITNeY Suite has been used. One of these is the case from [T4], which is described in Sect. 3.3.

One observation we can make is that most of the visualisations tools from the previous section are integrated very tightly with the editor of some modelling formalism or are low-level libraries that have to be integrated into real tools. This means that it is difficult or even impossible to use the visualisation tools with other formalisms. Furthermore, all of the tools except SceneBeans have a closed architecture, which makes it difficult to extend the functionality of the tools for people other than the formal method developers. In Table 3.1 the five visualisation tools mentioned in the previous section have been compared with the BRITNeY Suite. We note that the BRITNeY Suite is shown in two different variations, BRITNeY Suite 1 and BRITNeY Suite 2. The version of the BRITNeY Suite described in this section, [T3], and used in the case study of [T4] is labelled BRITNeY Suite 1. BRITNeY Suite 2 is the label of the version of the BRITNeY Suite described in Sect. 3.4 and [T5]. The column "Tool/formalism independent" shows whether the visualisation tool is tied so close to a modelling tool that it is impossible to use it independently of the tool. The column "User extensible" shows whether it is possible to extend the tool for people other than the original developers. "Standard widgets" and "User-drawn visualisation" show whether the tool supports standard widgets like check-boxes, gauges, and buttons or animated cartoons or drawings specified by the user. "GUI for creating visualisations" shows whether the visualisations can be drawn by the user using a user-friendly designer. "Dynamic instantiation of objects in visualisation" shows whether it is possible to instantiate objects in the visualisation (as opposed to requiring that all used objects must be created manually before starting visualisation). The columns "Synchronous operation" and "Asynchronous operation" indicate whether the tool supports halting the execution of the model and waiting for user input respectively if it



(a)



(b)

Figure 3.4: The model of a network protocol from Fig. 1.5 with annotations to drive a visualisation (a) and a resulting visualisation (b).

is possible for the user to manipulate a running simulation without requiring that the model stops and waits for input. “Integrated with formalism” indicates whether the tools supports a natural binding of the visualisation to the model.

Table 3.1: Comparison of various visualisation tools.

Tool	Tool/formalism independent	User extensible	Standard widgets	User-drawn visualisations	GUI for creating visualisations	Dynamic instantiation of objects in visualisation	Synchronous operation	Asynchronous operation	Integrated with formalism
ExSpect			✓		✓			✓	✓
MIMIC/CPN		✓[a]		✓	✓	✓	✓		
LTSA + SceneBeans	✓[b]	✓		✓			✓	✓	✓
PNVis				✓	✓[c]	✓	✓	✓	✓
Play-Engine			✓				✓	✓	✓
BRITNeY Suite 1[d]	✓	✓	✓	✓	✓[e]	✓	✓		
BRITNeY Suite 2[d]	✓	✓	✓	✓	✓[e]	✓	✓	✓	✓

[a] MIMIC/CPN can be extended using SML code, but this is not for the faint of heart.

[b] The SceneBeans library can be used independently of LTSA, but must be integrated in a Java program.

[c] Object description are created using standard VRML files and they can be created using most 3D drawing programs. The description of the world must be written manually as an XML file.

[d] The version of the BRITNeY Suite presented in [T3] (BRITNeY Suite 1) did not have support for asynchronous operation and formalism integration. The version described in [T5] (BRITNeY Suite 2) does support this.

[e] Depends on the visualisation.

3.3 Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks [T4]

The paper [T4], co-authored by the author of this thesis, describes an industrial case study where coloured Petri nets have been used to develop a formal model and a model-driven prototype of a network protocol. The project [101] is a collaboration between Ericsson Denmark A/S, Telebit [47] and the CPN group at the University of Aarhus [34].

In Figs. 3.5 and 3.6, we see two visualisations of an interoperability protocol for mobile ad-hoc networks [T4]. The protocol is used to ensure that the mobile ad-hoc nodes (the laptops) can communicate with the stationary host, even when on the move. Each gateway owns a specific sub-net of IP addresses. Based on the IP address of an ad-hoc node, it is possible to decide which gate-

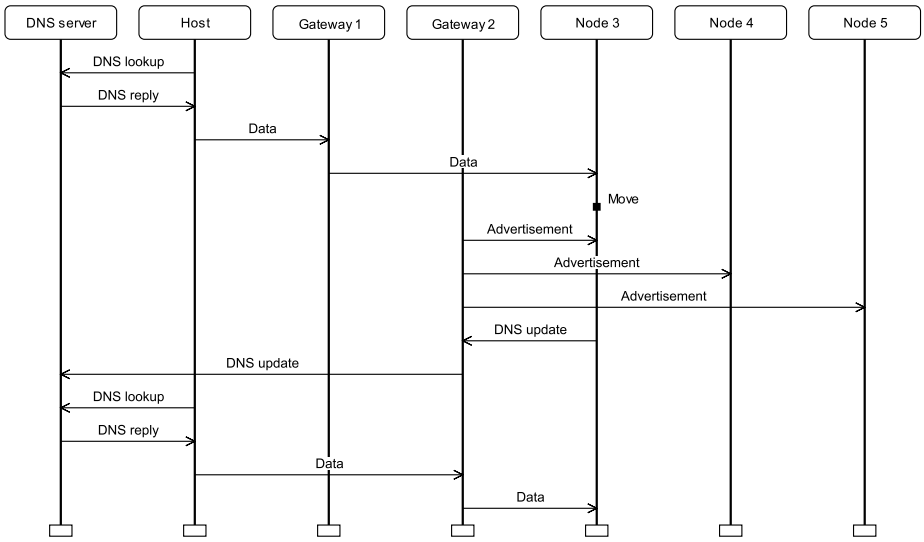


Figure 3.5: A visualisation of an interoperability protocol for mobile ad-hoc networks using message sequence charts.

way to use. The basic operation of the model is illustrated by the message sequence chart in Fig. 3.5. When a Host wants to transmit data to a mobile node, say Node 3, it looks up its address at the DNS server, which returns the IP address of the mobile node. From the IP address, the Host knows to send the packet via Gateway 1, which is closest to the mobile computer. The gateway forwards the packet to Node 3. Now, Node 3 moves physically, leading to it being closer to Gateway 2. Now, at some point in time, Gateway 2 sends out a gateway advertisement to all reachable mobile nodes. When Node 3 receives the advertisement it discovers that it is closer than Gateway 1. Node 3 switches IP addresses to one in the prefix owned by Gateway 2 and transmits a DNS update, via its new gateway, to the DNS server. If the host now wants to send data to Node 3, it will receive a new IP address from the DNS server, and conclude that packets to Node 3 should now go though Gateway 2. The visualisation in Fig. 3.6 enables users to observe the behaviour of the system as coloured dots, representing packets, flow along the network. Furthermore, the visualisation allows users to provide stimuli to the protocol by dragging and dropping the laptops to indicate node movement. The use of an underlying formal model can be completely hidden when experimenting with the prototype. The domain-specific GUI has been used in the project both internally during protocol design and externally when presenting the designed protocol to management and protocol engineers not familiar with CPN modelling. The message sequence chart in Fig. 3.5 is also created using the BRITNeY Suite.

In this project the goal was not to arrive at an implementation but rather to evaluate different techniques to facilitating communication between stationary hosts and mobile nodes which may move during communication. This means that the visualisation and formal model was actually the product rather than a means to construct correct software. A contribution of the paper was therefore the idea of using the method in Fig. 1.3 to produce a model-driven prototype. Our industrial partners, Ericsson Denmark A/S, Telebit, in parallel with the implementation of the model-driven prototype made an implementation of a simpler version of the protocol using real software and hardware, and the

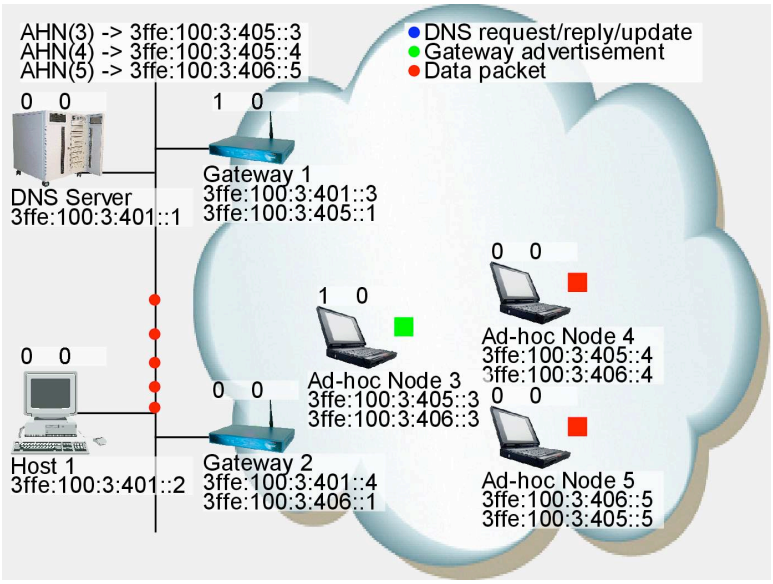


Figure 3.6: A cartoon-like visualisation of an interoperability protocol for mobile ad-hoc networks based on network diagrams.

two prototypes were both presented to management. The model-driven prototype has several advantages over a hardware based prototype, including that a model-based prototype is easier to control compared to a physical prototype, in particular in the case of mobile nodes and wireless communication where scenarios can be very difficult to control and reproduce. Furthermore, implementation details can be abstracted away and only the key parts of the design have to be specified in detail. As an example, in the CPN model of the interoperability protocol we have abstracted away the routing mechanisms in the core and ad-hoc networks, the mechanism used for distribution of advertisements, and how nodes determine distance to gateways. Instead, we have modelled the service provided by these components only. The possibility of making abstraction means that it is possible to obtain an executable prototype without implementing all components. Also, the use of a model means that there is no need to invest in physical equipment and no need to set up actual physical equipment. This also makes it possible to investigate larger scenarios, e.g., scenarios that may not be feasible to investigate with the available physical equipment. All of these advantages stem from the fact that we have created an abstract formal model. It would have been difficult to present the model to management and engineers without a visualisation, however. The use of visualisation on top of a formal model yields further advantages, including that the behaviour observed by the user is as defined by the underlying model that formally specifies the design. The alternative would have been to implement a separate visualisation package in, e.g., Java, totally detached from the CPN model. We would then have obtained a model closer to the actual implementation, but the disadvantage of this approach would have been a double representation of the dynamics of the interoperability protocol. The use of a domain specific graphical user interface (the visualisation) has the advantage that the design can be experimented with and explored without having knowledge of the CPN modelling language. This is also illustrated by the fact that the idea of the protocol has been described in this section using the visualisations developed during the project. The work presented in [T4] has demonstrated that

using CP-nets and the supporting computer tools for building a model-based prototype can provide a viable and useful alternative to building a physical prototype. Furthermore, the CPN model can also serve as a basis for further development of the interoperability protocol, e.g., by refining the modelling of the routing and advertisement distribution mechanisms to the concrete protocols that would be required to implement the solution. There is still a gap from the CPN model to the actual implementation of the interoperability protocol, but the CPN modelling has yielded an executable prototype that can be used to explore the solution and serve as a basis for the later implementation.

Important lessons from the project include that asynchronous input to the model is important. The model-based prototype described in [T4] let the protocol perform actions itself, such as sending out gateway advertisements, but should react immediately when the user wants to send data to a mobile node or when a mobile node is moved. As the project built on a version of the BRITNeY Suite, which tied visualisations to models using inscriptions (the version described in Sect. 3.2), we had to implement polling of the visualisation for user interaction. When the model was able to perform a lot of transitions itself (e.g. when a lot of packets and gateway advertisements were outstanding), this would result in very poor feedback from the visualisation. Additionally, the verbose inscriptions required to keep the visualisation up-to-date made it difficult to describe the model to the engineers who actually had experience with CPN models.

3.4 A Game-theoretic Approach to Behavioural Visualisation [T5]

As can be seen from Table 3.1, the version of the BRITNeY Suite described in Sect. 3.2 did not have support for asynchronous operation and integration with the formalism. The tool, as described in Sect. 3.2 and in [T3], is only able to support synchronous operation as visualisation functions are called whenever a transition occurs and it is not possible to do the opposite: force a transition to occur whenever something happens in the visualisation. This is acceptable if the purpose of the visualisation is only to show the operation of the models, such as the MSC in Fig. 3.4(b), which shows the execution of the network protocol in Fig. 3.4(a), but did, e.g., not suffice for the visualisation in Fig. 3.6, where the model should perform tasks in the background and react immediately on user stimulation such as when a node is moved. Another problem is that the added annotations are big and hardly declarative, which clutters the model and makes even the simple models seem complex as illustrated by the annotated model in Fig. 3.4(a). This can be alleviated by using Lindstrøm and Wells' monitors [113], which basically move the inscriptions to a separate list. A disadvantage of this approach is that the inscriptions are merely hidden, which makes it difficult to see the connection between the visualisation and the formal model.

While it is possible to ask the user for very simple information if we accept stopping the execution of the model meanwhile, if we want to create a visualisation which shows the operation of the model while allowing the user to stimulate the model, we will need to implement a polling mechanism. This clutters the model even further and makes the model and visualisation seem unresponsive as the model will not react until the visualisation is polled by the model. Furthermore, as annotations have to be added to each transition, it is easy to forget some. If, e.g., the inscription at the Drop transition in Fig. 3.4(a)

is omitted, we would never see the Drop event on the Network. This may lead domain experts to believe that packets cannot be lost. Additionally, this way of adding visualisations is unique to CPN models (though the idea of executing code whenever a transition is executed of course can be adapted to other formalisms as well). Finally, using this approach makes it difficult to switch visualisations on and off unless we use monitors, which can be switched on and off individually. For example, when we do analysis using the reachability graph method as described in the previous chapter, we will need to execute a lot of transitions. As transitions may not be executed in the order they would during a simulation of the model, the resulting visualisation will often be useless and only slow down analysis. It is also possible that we may wish to create more than one visualisation for each model, for example we may want to create a visualisation like the one in Fig. 1.13 and one like Fig. 3.4(b) for the network protocol model in Fig. 3.4(a). If we have more than one visualisation we may only want to see the result of the execution of the model using one visualisation or we may want to see the result in both. As we are allowed to execute arbitrary code when a transition is executed, it is of course possible to write code that facilitates this, but it will hardly be easy to read and modify and therefore difficult to maintain.

The paper [T5] defines a formal framework for visualisations, which tries to alleviate these problems, as we shall see later. The idea is to view a visualisation as a formal model and synchronise it with the formal model we want to visualise. To make it possible to view the result of the execution of the formal model as well as provide stimulation to the model without letting the visualisation change the behaviour of the model, we rely on the notion of games. A game is basically a labelled transitions system where the transitions are partitioned into controllable and uncontrollable transitions. The notion is a formalisation of normal board games, such as tic-tac-toe. Here a player plays against an opponent. The player is able to make certain moves (such as drawing a cross on the board) whereas the opponent is able to make other moves (such as adding a nought to the board). The player is able to decide which moves he wishes to make, his moves are controllable, whereas he is incapable of controlling which moves the opponent wishes to make, they are uncontrollable. This is formalised in Def. 3.1.

Definition 3.1 (Game) *A **game** (or game transition system) is a tuple,* $\mathcal{G} = (S, T^u, T^c, \Delta, s_I, W)$*, where*

- $S \neq \emptyset$ *is a set of **states**,*
- T^u *and* T^c *are sets of **uncontrollable transitions** respectively **controllable transitions** such that* $T^u \cap T^c = \emptyset$*,*
- $\Delta \subseteq S \times (T^u \cup T^c) \times S$ *is the **transition relation** indicating **successor states**,*
- $s_I \in S$ *is the **initial state**, and*
- $W \subseteq S$ *is a set of **winning states**.*

A similar definition can be created for any formalism which uses transition systems as semantical foundation. One example of such a formalism is coloured Petri nets, which can be extended to *game coloured Petri nets*, introduced in [C4], by the author of this thesis. Game coloured Petri nets are coloured Petri nets except the transitions are separated into controllable and uncontrollable ones. Consider, e.g., the model in Fig. 3.7. This is the same model as the one in Fig. 1.5 except the Drop transition is drawn with a dashed line. This

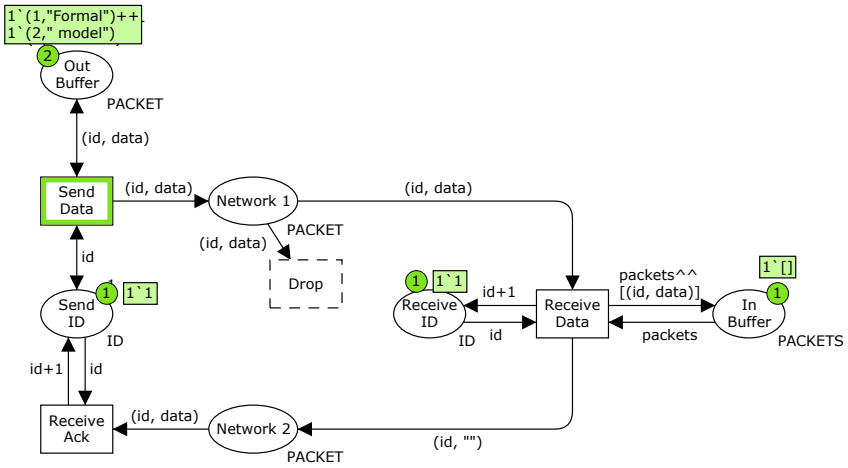


Figure 3.7: A formal model of a simple protocol modelled as a game coloured Petri net.

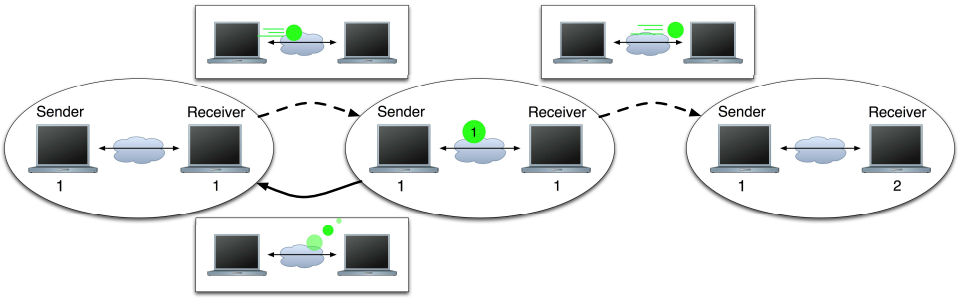


Figure 3.8: Fragment of a visualisation of a simple network protocol as a labelled transition system.

indicates that the Drop transition is uncontrollable. The rest of the transitions are controllable. In this way we state that the network protocol is only able to control what happens at the sender and receiver, it is unable to control whether the network drops packets. In this way we have modelled the system we wish to analyse, we have modelled the environment (the network), and we have modelled how the two can interact.

The idea of the work in [T5] is to view visualisations as game transition systems synchronised [2] with formal models also modelled as game transition systems. The visualisation plays one side of the game and the formal model the other. The rationale behind the idea of considering visualisations as transition systems is that we can consider what is visible in the visualisation as a state of the visualisation, and changes to what can be seen can be considered as transitions of the visualisation. Consider for example the fragment of a visualisation of the simple network protocol in Fig. 3.8. Here we see three states (the ovals) and three labelled transitions between them (the arrows with labels in rectangles). The states show different static views and the transitions are animations taking the visualisation from one state to the next. Some of the arrows are dashed, indicating that they are uncontrollable, i.e., not controlled by the environment but by the system we model.

If we allow all synchronisations between a visualisation and a model, the

behaviour of the synchronisation is not defined by the model, but by the model and the visualisation in unison. As an example, if we create a visualisation consisting of only one state and no transitions, the synchronisation between this and any formal model is also without behaviour which is not what we want to obtain, as this corresponds to creating a visualisation of the network protocol in Fig. 3.4(a) and omitting a visualisation of the Drop transition. We want the behaviour of the synchronised system to be dictated by the model and use the visualisation to show what happens in the model (we will deal with stimulation of the model shortly). In order to do this, we require that the visualisation is able to simulate [124] the model. In that way, the behaviour of the synchronisation is dictated entirely by the model.

If we also want to manipulate the execution of the model, we need to loosen the requirement that the visualisation must be able to simulate the model. Rather than allowing arbitrary synchronisations, which would make it difficult to distinguish between actions taken by the model itself and actions initiated by the user, we rely on games. The idea is that the visualisation plays one side of a game and the model plays the other side; controllable transitions of the visualisation are executed synchronised with uncontrollable transitions of the model and vice versa. We require that the uncontrollable transitions of one side can simulate the controllable transitions of the other side. This is formulated in Def. 3.2.

Definition 3.2 (Visualisation) *Given a model as a game* $\mathcal{G}_M = (S_M, T_M^u, T_M^c, \Delta_M, s_{IM}, W_M)$*, a* ***visualisation*** $\mathcal{G}_V = (S_V, T_V^u, T_V^c, \Delta_V, s_{IV}, W_V)$*, and a* ***synchronisation constraint*** $\mathcal{S} \subseteq (T_M^u \times T_V^c) \cup (T_M^c \times T_V^u)$*, we say that* $\mathcal{G}_V$ *can be used as a visualisation of* $\mathcal{G}_M$ *with* $\mathcal{S}$ *iff there exists a relation* $\sim \subseteq S_M \times S_V$ *such that whenever* $s_M \sim s_V$

- *for all* $\alpha \in T_M^c$ *if* $s_M \xrightarrow{\alpha} s_M'$ *there exist* $s_V' \in S_V, \beta \in T_V^u$ *such that* $s_M' \sim s_V'$*,* $(\alpha, \beta) \in \mathcal{S}$*, and* $s_V \xrightarrow{\beta} s_V'$*, and*
- *for all* $\beta \in T_V^c$ *if* $s_V \xrightarrow{\beta} s_V'$ *there exist* $s_M' \in S_M, \alpha \in T_M^u$ *such that* $s_M' \sim s_V'$*,* $(\alpha, \beta) \in \mathcal{S}$*, and* $s_M \xrightarrow{\beta} s_M'$*.*

Furthermore we require that $s_{IM} \sim s_{IV}$*.*

In [T5] two example visualisations are described: message sequence charts and SceneBeans visualisations. These visualisations correspond to Fig. 3.4(b) and Fig. 1.13 respectively.

This view of visualisations addresses all the aforementioned problems: Tying the visualisation to a model no longer requires inscriptions, but rather the definition of a synchronisation constraint. This constraint can of course be defined using inscriptions, but it can also be specified separately, like in the case of monitors. All the visualisations described in [T5] and [C4] comes with a constraint which uses conventions rather than specifications to synchronise visualisations to CPN models, completely eliminating the need to manually tie visualisations to formal models. The basic idea is to use the name of transitions and places to tie the visualisations to models. For example, when a transition named Send Data is executed, an event named sendData can be generated in a MSC, or the user can be shown a dialog box titled Send Data. In addition to reducing clutter, this approach to visualisation also makes it possible to turn visualisations on and off easily, as we can just state that a certain constraint and visualisation should not be used during execution of the model. As definition 3.2 requires that whenever the formal models makes a controllable move, the visualisation must be able to make a corresponding uncontrollable move, it becomes impossible to forget an inscription leading to erroneous visualisations. If

we wish to ignore an event form the model, we must do so explicitly by adding a transition to the visualisation corresponding to "do nothing" and synchronising this transition with the event we wish to ignore. Furthermore, the definition comes with built-in support for both synchronous and asynchronous operation. While the definition requires that the visualisation and formal model always run synchronously, it allows information to flow in both directions. Here information flow from the formal model to the visualisation corresponds to visual updates, whereas flow in the other direction corresponds to stimulation of the model. This corresponds to asynchronous operation: when the user has provided no input, the model just executes normally and as soon as the user provides input the model is able to receive that input. Synchronous operation is done simply by letting the model execute no transitions, e.g., because by letting only controllable transitions (from the point of view of the model) be enabled.

The paper [T5] additionally considers implementation details. Firstly, a Java interface is presented. The interface makes it possible to implement visualisations without knowing which formalism will make use of it. As long as a formalism has a semantical foundation in games it is possible to immediately synchronise models created using the formalism with visualisations without changing the formalism. The paper also considers how to deal with fairness when executing a formal model with a visualisation. For example, we may often want the formal model to react immediately on user input, and this can be done by giving priority to uncontrollable transitions (from the point of view of the formal model). Additional fairness criteria, such as delays and strict alternation are also considered. Two example uses of visualisations are given: a revised version of the industrial case study described in [T4] and Sect. 3.3 and visualisation of a certain kind of properties of reachability graphs.

3.5 Contributions and Future Work

In this chapter we have discussed a number of tools that can be used for visualisation of formal models. In particular we have seen the BRITNeY Suite, which is developed by the author of this thesis. We have considered the architecture of the BRITNeY Suite and we have seen an industrial case study where the BRITNeY Suite has been used to develop a model-based prototype of a protocol facilitating communication between mobile nodes in ad-hoc networks. Finally, we have seen a game-theory-founded formal framework for describing visualisations which gives visualisations a formal semantics, and provides a foundation for tying visualisations to formalisms without altering either. This section will discuss the contribution made to visualisation of formal models and some applications and experiences by the author of this thesis. We go on to describe several applications of the BRITNeY Suite by other research groups, and finally we provide some directions for future work.

The BRITNeY Suite, as presented in [T3], provides a tool which makes it possible to visualise formal models. The tool is extensible by means of plug-ins and has been integrated with CPN Tools. The BRITNeY Suite has already been used in several projects, among these a project to to build a model-based prototype of a network protocol, as described in out paper [T4]. This project heavily influenced the development of the BRITNeY Suite as it suggested several possible improvements of the BRITNeY Suite. One problem we observed was that at the end of the project our industrial partner, Ericsson Denmark A/S, Telebit, would like a copy of the developed prototype. Distribution of the prototype was easy enough, but a very brief manual to help starting the prototype grew extremely long because setting up the prototype for experimentation was rather

complex. Therefore the BRITNeY Suite was changed to allow web-start [95] launch of visualisations. By writing a simple specification and designing the CPN model in a certain way (the details are available in our workshop paper [C6]), it is possible to upload the BRITNeY Suite to a web-server and allow users to start the visualisation using a single click in a web-browser.

Two other problems encountered in the industrial case study from [T4] have been alleviated. Firstly, we needed to be able to stimulate the model during simulation, and secondly, the very verbose annotations to the model made a relatively easy to understand model seem overly complex. In the paper [T3] we suggest that asynchronous interaction between the formal model and the BRITNeY Suite could happen via special *fusion places* [91, Chap. 3], and in [C3] we suggest that synchronous channels between the formal model and the visualisation could alleviate the need for complex annotations of the model. We believe that the direction taken in [T5], where visualisations are regarded as games synchronised with the formal model, is nicer, more declarative, and more formalism independent. In addition to the message sequence chart and cartoon-line Scenebeans visualisations based on this idea, both presented in Sect. 3.4 and [T5], our paper [C4], gives a third, CP-net specific, example of a visualisation. This visualisation makes it possible to automatically generate form-filling applications from a CPN model.

3.5.1 Applications by the Author of this Thesis

In addition to the case study described in [T4] and Sect. 3.3, we have used the BRITNeY Suite in various other settings. Some of these will be described here.

The BRITNeY Suite Platform for Experiments with Coloured Petri Nets

In [C5], we extend the scope of the BRITNeY Suite, by showing how it is possible to use the BRITNeY Suite to experiment with the CPN formalism. Thereby we broaden the audience from formal methods experts, developing and visualising formal models, to also include formalism developers, who improve the formalism. This is possible by using the pluggable architecture of the BRITNeY Suite to extend the tool and use fairly high-level constructs to interact with the CPN model. It is possible use this to make high-level experiments with the formalism. Some formalism developers think of new constructs whose purpose is to make it easier and more natural to use the CPN formalism. Such extensions include transition fusion [22] (or synchronous channels), inhibitor arcs [21], bounded places, FIFO (first-in-first-out) places, and prioritised transitions. All of these constructs can be given a semantics by simply translating them to regular CP-nets. Using the scripting facilities described in [C5], we show how to implement a custom scheduler, which makes it possible to prioritise transitions in as little as 30 lines of code, demonstrating that it is relatively easy to implement support for new language constructs to validating whether they are useful.

Command-line loading of CPN models

The BRITNeY Suite has also been used by the author of this thesis in a more unconventional way, namely to load CPN models from the command line. In the ASCoVeCo project (Advanced State Space Methods and Computer tools for Verification of Communication Protocols) [3] at the University of Aarhus, among other things, an automated test-suite of a tool for reachability graph analysis

is being developed (for more details refer to Sect. 2.5.1). The author of this thesis participates in the ASCoVeCo project. As the tool implements reachability graph analysis of CPN models, it is necessary to automatically compile the tool, load a model, and run analysis in order to automatically test the tool. The trouble arises when we want to load the model, as CPN Tools [C1, 33] has no means to do that from the command-line. Furthermore, as the CPN Tools editor does not use the Model-View-Controller design pattern from Fig. 1.10, it would be difficult and tedious to implement this feature.

As can be seen in Fig. 3.3, CPN Tools actually consists of two separate components, an editor and a simulator. The simulator is only able to communicate with one process at a time, so Fig. 3.3, while conceptually correct, does not actually reflect how communication takes place in practise. The BRITNeY Suite generates and injects stubs into the CPN simulator, and therefore needs to communicate with the simulator, so it implements a proxy, which mediates the communication from the CPN editor to the simulator. Exploiting this proxy, it is easy to record and replay this communication between the CPN editor and the simulator using the BRITNeY Suite and later replay it. The BRITNeY Suite implements the Model-View-Controller design pattern so it is easy to create a command-line version which is able to replay the recorded communication.

Automatic testing thus consists of first recording the communication between the CPN editor and the simulator for each model in the test-suite. This step requires manual intervention to load the model using CPN Tools, but only has to be done once to generate a recording. Now, each time we wish to run a test, we just need to recompile the reachability graph analysis tool (this of course only has to be done once for each test run; after that the result can be re-used for all models) and load the model by replaying the recording using a command-line version of the BRITNeY Suite. Finally, we load and run the test.

Exploiting the BRITNeY Suite in this manner made it possible to implement automatic loading of CPN models into the simulator in days rather than weeks or months, which would be required to create a loader from scratch or to refactor CPN Tools to make it possible to create a command-line version.

3.5.2 Applications by other Research Groups

All applications discussed until now has been made by or in cooperation with the author of this thesis. The BRITNeY Suite has also been used by several other individuals and research groups. Use ranges from simple visualisation of formal models, which is of course the main application of the BRITNeY Suite, over meta-visualisation, where the BRITNeY Suite is used to provide visualisation of other formalisms by translating formal models into coloured Petri net models, to other applications, where the BRITNeY Suite is used in nontraditional ways to, e.g., to integrate the CPN simulator into a multi-formalism tool. In this section we will provide some examples of applications BRITNeY Suite. We will only provide few examples from the first category as the idea of such applications is often very similar to our own application in [T4]. Some of these applications are not published yet due to the fact that the BRITNeY Suite was released to a broad audience in September 2006, only nine months before this overview paper was written. Thus some of the applications described are only known to the author thanks to personal communication. In these cases no publications are cited, but the name and affiliation of the contact person is mentioned.

Visualisation of blanc-loan applications

In [94] Jørgensen and Lassen use the BRITNeY Suite to create a visualisation for requirements engineering of a new workflow system [164] for banks. The goal of the workflow system is to support the handling of blanc loan applications. Users can interact with the visualisation by, e.g., setting up loans for customers to make a loan request, or by changing the status of loan requests on behalf of bank assistants and a bank manager to, e.g., grant or reject the requests. The use of an abstract visualisation allows users to focus on the workflow and not on how the interface of the future system should look like.

Visualisation of electronic patient record

In [144] Jørgensen, Lassen, and Aalst present a use-case consisting of a electronic patient record to be developed for Fyns County in Denmark. The work builds on task descriptions, corresponding to the specification in Fig. 1.3, which are translated to a model of the problem using coloured Petri nets. This model is visualised using the BRITNeY Suite in order to validate that it really corresponds to the task descriptions (specification) using the approach in Fig. 1.3. From the coloured Petri net model a model of the system is constructed in Aalst, Jørgensen, and Lassen's coloured workflow nets [162] and translated to Aalst and Hofstede's YAWL (yet another workflow language) [163], which is an executable workflow language.

Visualisation of behaviour of UML sequence diagrams

In [114] Machade et al. consider the derivation of *system requirements* from *user requirements*. User requirements are requirements for a system imposed by the future users of the system, and system requirements are requirements from the developers, which makes is possible to satisfy the user requirements in an implementation. Basically, [114] deals with going from the specification to the formal model in Fig. 1.3.

The formal model (system requirements) is assumed to be specified using UML [131] sequence diagrams, and the authors wish to use a method similar to the one in Fig. 1.3 to validate that the formal model corresponds to the specification (user requirements). To do that, UML sequence diagrams are translated into CPN models and the BRITNeY Suite is used to visualise their behaviour. This is thus an example of a meta-visualisation, as the BRITNeY Suite is used to provide visualisations of formal models created using sequence diagrams. The approach is exemplified using an information system called uPAIN whose main concern is pain control of patients in a hospital.

In [145], Ribeiro and Fernandes also consider translation of UML sequence diagrams to CPN models in order to facilitate visualisation of UML sequence diagrams. Here a case study of an industrial reactor system is presented.

Implementation of a workflow simulator

When implementing workflow systems, one typically uses a language or tool designed specifically for this. One advanced example of such a language is YAWL [163]. Using YAWL it is possible to automatically generate a user interface, which makes it possible for participants to acquire and complete tasks. Another workflow language is coloured workflow nets [162], which are a restricted form of coloured Petri nets. In order to obtain automatic generation of a visualisation of the workflow system, the BRITNeY Suite is used. This

work is conducted by Kristian Bisgaard Lassen at the University of Aarhus, Denmark.

It is of course immediately possible to use the BRITNeY Suite for visualisation as coloured workflow nets form a sub-class of coloured Petri nets and hence can be executed by CPN Tools [C1, 33]. But, due to the fact that coloured workflow nets are restricted and have a quite predictable structure (which shall not be explained in this thesis), it is possible to generate a single visualisation which can automatically be used for *any* coloured workflow net model. The work uses the idea of regarding visualisations as games, and extends coloured workflow nets slightly by separating transitions into controllable and uncontrollable transitions. This makes models special cases of game coloured Petri nets [C4] models, which can be visualised by the BRITNeY Suite. Controllable actions are performed automatically by the workflow system, and uncontrollable actions must be performed by the user. The goal is to make visualisation of coloured workflow net models a push-button technology.

Delegation of complex calculations in CPN models to Java

While Standard ML is well-suited for functional calculations, such as the implementation of a coloured Petri net simulator, it is not very well-suited for calculations which depend on and update complex data structures. Furthermore, Standard ML is not as well-known as more main-stream languages such as Java, making it difficult to obtain off-the-shelf libraries for performing standard calculations or to employ programmers capable of implementing calculations. It may therefore be interesting to be able to perform some tasks, such as complex imperative algorithms, in Java rather than in Standard ML.

The plug-in architecture of the BRITNeY Suite was originally tailored for adding new kinds of visualisations, but can be used to call arbitrary Java code from CPN models. The first use of this is in [T4], where a plug-in named DataStore is developed. The plug-in makes it possible to maintain a set of counters, which in the project is used to show the size of the ingoing and outgoing buffers of nodes in a network in a visualisation. Riahi Bilel from Faculté des Sciences de Tunis (FST) uses the BRITNeY Suite to implement a protocol for sensor networks. The goal of the protocol is to conserve energy in the sensor network by turning off sensors not required for correct operation, i.e., that the entire area where the network is deployed is covered and that all sensors are able to communicate with a fixed base station. The protocol keeps track of a large number of sensors. A CPN model keeps tracks of the sensors and a plug-in, written in Java, calculates which sensors to turn on or off depending the sensor configuration. The plug-in consists of 5 Java classes and 1300 lines of Java code, which would be difficult to write in Standard ML unless the programmer is experienced in the language.

Integration of CPN simulator into multi-formalism tool

György Balogh from Vanderbilt University, USA, wants to integrate the CPN simulator into Morse et al.'s HLA (High Level Architecture) [84, 128], an interface for integrating simulation engines of different formalisms into a single tool. In order to facilitate such integration, the CPN simulator (or some glue code) must alert HLA whenever it wants to increase its global clock (using a version of CP-nets with support for time) from t_1 to t_2. HLA then makes a call-back when the CPN simulator is allowed to increase the time to t_2. HLA can also perform information exchange (add tokens to the model) and grant a smaller time increase $t_3 < t_2$ if the produced token is available at time t_3.

As indicated in Fig. 3.3 it is possible for external processes to communicate with the CPN simulator, but the protocol used [35] is quite complex and tedious to implement. Rather than implementing the protocol from scratch, the implementation which is a part of the BRITNeY Suite can be used. The BRITNeY Suite provides two levels of abstractions of the protocol. One makes it possible to exchange packets with the simulator. These packets must be constructed by the implementer and the interface takes care of translating an abstract description of packets into binary data. A higher level of interaction is also possible. Here a remote procedure call protocol is implemented on top of the interface for exchanging packets. Rather than worrying about constructing packets correctly, the implementer only has to construct an object-oriented representation of the model and use high-level method calls to interact with the simulator. It is even possible to use the CPN editor part of CPN Tools for loading the CPN model by using the simulator proxy or recording facility of the BRITNeY Suite as described earlier.

3.5.3 Future Work

In this section we will provide some directions for future work. As most of the ideas described earlier in this chapter has already been implemented and tested in practise, future work mainly consists of improvement of the tools and documentation. We also describe an interesting way to combine visualisations, as described in this chapter, with formal verification as described in Chapter 2, by using a visualisation to convey the fact that certain properties do not hold for a formal model.

Improvement of the BRITNeY Suite platform for experiments

As can be seen from some of the applications, the users of the BRITNeY Suite has broadened from consisting of formal methods experts wishing to visualise the behaviour of a formal model to also include formal methods developers experimenting with the formalism to evaluate extensions or to use the formalism in new ways. As described earlier and in a workshop paper by the author of this thesis, [C5], this is possible with the current version of the BRITNeY Suite thanks to a pluggable architecture and extensive support for scripting. This platform can be enhanced in several ways, however, and here we describe some useful improvements.

While CPN Tools supports incremental syntax check of CPN models, this is not supported by the BRITNeY Suite. This makes the BRITNeY Suite less usable for experiments, as time must be spent re-checking models from scratch. The current implementation automatically updates graphical representations of models as the internal representation is constructed, and ii would be nice to improve this to also support incremental syntax check, making it even easier to load, modify and experiment with CPN models.

Another, more pragmatic, problem is that while some examples exist, the documentation of the tool could be improved. Currently the documentation consists mainly of a couple of simple examples, which is fine for applications, which use the BRITNeY Suite for visualisation of concrete models. While examples exist that demonstrate how to create extension plug-ins (the source code for one is available in [C5]), they are very simplistic, and only show simple ways to interact with the internals of the BRITNeY Suite. This, of course, makes difficult to implement meta-visualisations or any of the unconventional applications of the BRITNeY Suite. Better and more advanced examples and a

better reference manual of the internals of the BRITNeY Suite would alleviate this problem.

Finally, some of the technical decisions made when the BRITNeY Suite was developed would probably be made differently today. The first change would be to use SOAP web-services [63] instead of XML-RPC [170] for invoking methods in extension plug-ins, and the second change would be to implement the BRITNeY Suite either as a plug-in to Eclipse [41] or as an Eclipse Rich Client Platform [120] application. Let us look at the advantages and disadvantages of each of these in turn.

The use of XML-RPC for communication with the extension plug-ins benefits from the fact the XML-RPC is an open protocol which is easy to understand and implement. Another way to communicate with remote programs is SOAP web-services, which, like XML-RPC, uses XML messages to invoke remote functions. SOAP web-services additionally supports the Web Service Definition Language (WSDL) [20], an XML-language for describing web-services. Using this language we could eliminate the stub-generator from the BRITNeY Suite, and make the clients (such as the CPN simulator) inspect the tool and generate stub-code themselves. This has the huge advantage that such clients exist for many programming languages, making it very easy to integrate support for the BRITNeY Suite in tools for simulation of formal models. The reason for not doing this already is that the SOAP web-services protocol is very complex and no client exist for Standard ML, the implementation language of the CPN simulator. As the CPN simulator is the primary user of the BRITNeY Suite, to not break this support and not implement a very complex client library, the BRITNeY Suite sticks with XML-RPC currently. Implementing support for SOAP in parallel with XML-RPC is being considered, however, in order to get the best of both worlds.

The BRITNeY Suite implements its own plug-in mechanism using a very simplistic plug-in library, the Java Plug-in Framework [89]. This makes it possible to load code on run-time, either from the local disk or from the Internet. The framework makes it difficult or even impossible to use an Integrated Development Environment (IDE) such as Eclipse for debugging and single-stepping through the application. If development had happened within Eclipse, building on the frameworks distributed with Eclipse, it would have been possible to debug the program within the Eclipse IDE. Furthermore, it would be possible to distribute plug-ins as Eclipse projects, enabling use of parts of the functionality without using the entire tool. Finally, it would be possible to immediately integrate the BRITNeY Suite into applications written using Eclipse's frameworks, thereby creating a single tool for writing real programs, creating formal models, and for visualising formal models (and possibly real programs as well). Work on moving the BRITNeY Suite to the Eclipse platform is currently started by the author of this thesis.

Improvement of implementation of visualisations as games

A prototype the framework based on game-theory has been implemented in the BRITNeY Suite. The prototype implements fairness of the execution, i.e., how control is transferred between the user and the tool, in a couple of ways, namely strict alternation and preference of uncontrollable (user initiated) transitions. While this is enough for simple examples, it would be very interesting to experiment with fairness defined by a timed formalism where the execution of transitions take time.

The current implementation focuses primarily on the events of the system. For example, the message sequence chart visualisation shows transitions only.

The SceneBeans visualisation is also only able to interact with the formal model via synchronised transitions. This is fine for formalisms that are primarily event-oriented, such as labelled transition systems, where states are opaque. In formalisms that are both state and event oriented, such as Petri nets and in particular coloured Petri nets, this is not satisfiable. For example, the visualisation developed in [T4], shown in Fig. 3.6, shows the contents of the DNS database (upper left corner), but this is not easy to do using the current implementation as all updates to the shown DNS database must be formulated as changes to the visualisation. It would be much easier to just state that the rectangle in the upper left corner should always reflect the contents of the place modelling the DNS database. The definition (Def. 3.2) allows this, as information can be exchanged via the synchronisation, but the implementation does not reflect that. It would be very useful to be able to declaratively reflect the state of he system in the visualisation.

Visualisation of error traces for property violations

This section assumes that the user is familiar with how winning strategies are calculated for games and how CTL properties are verified. While [T5], reprinted in Chapter 9, states that it is possible to visualise error traces to violations of properties discovered using reachability graph analysis, this has not been implemented and explored extensively.

We want to address the problem that is that it is very difficult to visualise the existence/non-existence of a winning strategy of a game. For games a winning strategy is basically an annotated reachability graph. All states where the user has a winning strategy are marked as such. Such an annotated graph can of course just be shown to the user with winning states coloured green and other stated coloured red. This can be useful to understand why no winning strategy exist for small examples. For large examples, such graphs can have an extremely large number of nodes, making such a visualisation useless in practise.

In [T5], we suggest using a visualisation of the system created using game-theory and let the user play against a winning strategy. The idea is that if a user needs conviction that a winning strategy exists, it is because he thinks he has a winning strategy for the other side. We let the user play according to his "winning" strategy – he plays by interacting with a visualisation of the formal model, while the tool makes moves according to the (real) winning strategy. As the tool knows a real winning strategy, the user will eventually arrive at a situation where the system performs some unanticipated action, which may convince him that no winning strategy exists. Otherwise, the user will believe he made a mistake, try again until he has exhausted all his options, and finally be convinced that the computer is always able to win the game. Unanticipated actions performed by the tool can be useful to understand why a winning strategy exists: if the action is allowed by the model but not by the specification, the model does not accurately reflect the specification. If, on the other hand, both the specification and the model allows the action, the specification may need to be modified and the model updated accordingly. This can be made to work because, as the user does not have a winning strategy in the initial state, the tool will just have to execute transitions ensuring that the user is never able to reach a state from which he has a winning strategy. This is possible because otherwise the user would have winning strategy.

In a similar way, we can let a user contend against the system to convince the user that a certain CTL property is not satisfied. A proof that a CTL formula does not hold is an annotated reachability graph, where the annotations

are sub-formulae of the CTL formula we wish to check. Each node of the reachability graph is annotated with all sub-formulae that hold in the corresponding state. A visualisation of the fact that the property does not hold also uses a visualisation of the model created as a game, and shows the user which formula he has to prove in the shown state, initially the entire formula. CTL formulae basically consist of statements that must hold on all traces reachable from a state and statements that must hold on at least one trace. The user provides a transition to execute (using the visualisation) whenever existence needs to be proved, and the tool chooses a transition whenever statements must hold for all traces (naturally selecting a trace where the property does not hold). Like in the case of visualising winning strategies of games, this will eventually lead to a situation where some atomic proposition does not hold for the current state, in which case the strategy of the user was incorrect, causing the user to accept that the property does not hold or to try again.

It would be nice to have an implementation of this idea in order to experimentally validate that it is a useful way to show error traces. Of course, a visualisation created in this way can also be used to show error traces for simple properties for invariant and LTL properties.