Looking Good, Behaving Well

Behavioural Verification and Visualisation of Formal Models of Concurrent Systems

Michael Westergaard

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Looking Good, Behaving Well

Behavioural Verification and Visualisation of Formal Models of Concurrent Systems

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

by
Michael Westergaard
July 24, 2007

Abstract

Computer systems are so complex and crucial to our lives that we need to verify that they are correct and do not fail or risk facing enormous economical consequences, like in the case of the European Space Agency's Ariane 5 rocket, which self-destructed 37 seconds after launch because of a software malfunction, or loss of human lives, like the Therac-25 radiation therapy machine, which caused at least six deaths due to overdoses of radiation because the machine was not able to detect a human error. We would like to reduce the number of such errors or even prove their absence.

Many errors stem from incomplete and inconsistent specifications of the systems to construct, as they are often written in natural language text. We would instead like to create a formal specification. In order to do that, we create a formal model of the system we wish to construct, much like how an architect creates a blueprint of a house that is to be constructed.

A specification, in the form of a formal model, can then be verified using formal analysis methods. One such method is the reachability graph method, which basically explores all possible executions of the formal model by creating a graph where each node is a state of the formal model and each edge indicates that it is possible to go from the source to the destination state. Such a graph is called a reachability graph. The reachability graph method has the advantage that is can be implemented in a computer and made almost completely automatic. Unfortunately, the behaviour of a formal model can be very complex, so we will often need a reduction technique, which tries to explore only part of the behaviour or represent the behaviour more efficiently in the computer memory. This thesis presents two such reduction techniques. One reduction technique, the sweep-line method, uses a user-specified notion of progress to remove states that will never be encountered again from memory. This method has the disadvantage that the structure of the reachability graph is not preserved, so certain properties cannot be verified. In order to overcome that, we have extended the sweep-line method to also store the structure of the reachability graph in a memory-wise nearly-optimal manner. Another method, the ComBack method, avoids storing the states of the reachability graph altogether, by exploiting that any state can be reconstructed if we know how to get to it from the initial state. By storing a spanning tree of the reachability graph, rooted in the initial state, the ComBack method manages to represent the reachability graph very efficiently.

While a specification written as a formal model makes it possible to verify desired properties, it is often difficult or even impossible for domain experts, who know about the system we wish to construct, to validate that the formal model indeed corresponds to the desired system. In order to facilitate communication of the formal model, we create a visualisation of the behaviour of the formal model. The behaviour of the visualisation is completely defined by the formal model, and the visualisation makes it possible to provide input to the formal model. This thesis presents three papers on this topic. One presents

a tool, the BRITNeY Suite, which makes it possible to create visualisations of formal models. Another paper describes an industrial case study where formal models and visualisations have been used to create a prototype of a network protocol facilitating communication between computers moving from one wireless network to another. The third paper provides a formal game-theoretic framework for tying visualisations to formal models.

This thesis deals with making formal models look good and behave well. By creating a domain specific visualisation, we can make the model look good, and allow domain experts to understand them. By verifying the model using the reachability graph method, we can make the model behave well, by removing errors in the model, making the model better suited as specification. This thesis consists of two parts. Part I gives an overview of formal models, their analysis, and visualisation of them. Additionally, Part I describes the five papers, which are re-printed in Part II. Four of these papers have been published at conferences and one is submitted to a workshop.

Acknowledgements

I would like to thank current and former members the CPN group at the University of Aarhus. I would like to thank Lars Kristensen and Kurt Jensen, my supervisors, and gravity for pulling me in the right direction during my PhD studies. Thanks to Kim G. Larsen for an inspiring stay in Aalborg. I would like to thank Thomas Mailund, Kristian B. Lassen, my other co-authors, and Søren Christensen for fruitful discussions and cooperation. Thanks to my girlfriend, Lea Pedersen, for support under the difficult times of writing this thesis. I would like to thank my fellow students and friends for being there when I needed them and my mortal enemies for not. A big thank you to Fredagscaféen, Universitetsbaren, Cockney Pub, TÅGEKAMMERET, and Fysisk Fredagsbar for support during beverage-critical times, and to Brygshoppen for selling fine Belgian beers. I would like to thank the Moon for being in just the right place to not screw up the tides and the Sun for not being bright enough to shine through the roof and making flares on my computer screen. Also, a big thank you to everybody I forgot, except for those I do not want to thank. To move from the latter to the first group buy me a beer. Finally, a big thank you to you, my reader, for bearing with me for the next 171 pages.

Michael Westergaard,
Århus, July 24, 2007.

Contents

Part I

Overview

Chapter 1

Introduction

Modern computer systems are very important to our lives. Use spans from space shuttles and robots investigating foreign planets over critical hospital systems, power-plant control and computer systems controlling aeroplanes and cars, to home banking, e-mail, and word processing.

Some of the systems using computers can be considered highly critical, either because they are very expensive to produce, such as robots investigating foreign planets, or because human lives depend on them, such as computers controlling aeroplanes and cars, and systems in hospitals and power-plants. These systems need to be correct, because their failure can cause loss of human lives or enormous economical losses. Faulty computer software has been the cause of numerous disasters [81]. Disasters range from killing at least six people due to radiation overdoses because the Therac-25 radiation therapy machine [109] was unable to detect a human error and issue a warning, over economical losses, including the Ariane 5 lifting rocket [39], which self-destructed because a 64 bit floating point value was erroneously converted to a 16 bit integer and the error handler, which was supposed to take care of errors when too large values were converted, had been disabled for efficiency reasons, causing the computer to crash. The Mars Climate Orbiter [48] crashed while trying to land due to a mix-up between metric and U.S. customary units, causing the loss of the robot. NASA satellite software designed to measure holes in the ozone layer [129] ignored values deviating from the expected values caused a hole in the ozone layer to be ignored for eight years. These examples illustrate how errors in computer software can lead to vast economical losses, environmental disasters, and loss of human lives, so for critical software it is worth the effort to ensure that the software has no errors.

The goal of this work is to contribute to the design and improvement of methods for avoiding such catastrophic computer malfunctions. This is done by constructing a formal model of the system we wish to construct, validate that the formal model corresponds to the intended system using a domain-specific visualisation, and formally verify that the formal model satisfies properties required of the system, e.g., that it is impossible for a human error to cause the death of other humans. Correctness depends on the computers themselves, the hardware, and the programs they run, the software. Both are equally important and need to be correct in order for the entire system to be correct. In this thesis we focus on the correctness of the software, not because the hardware is deemed less important, but because we assume that somebody else takes care of the correctness of hardware.

In the rest of this chapter, we first describe what we consider formal models of computer systems. We then turn to describing how to verify the behaviour

of formal models and how to visualise that behaviour. After that we provide directions for arriving at a correct implementation from the formal model, and we end this chapter by providing a guide to the rest of this thesis.

1.1 Approaches to Software Validation

One classical approach to program correctness is to annotate them with formal expressions, e.g., using Hoare logic [70], from which we can derive desired correctness properties. We must manually or semi-automatically prove that the program indeed satisfies these annotations. This approach has a number of disadvantages. Firstly, the approach requires that humans manually find the correct annotations and prove their correctness. This is a lot of manual work, which is very difficult for large systems. Secondly, the approach requires that whenever a change is made to the system, some expressions must be re-evaluated and re-proven, which makes changes expensive.

Given the limitation of classical approaches to correctness, we instead consider an approach based on construction of models. To illustrate this, we consider a parallel in the physical world, namely building a house. When a customer wants a new house, but is unable to build it himself, he hires some contractors to do it for him. The customer here corresponds to the customer who wants a computer program, and the contractors correspond to software companies or programmers. The customer can send requirements (such as size, number, and placement of rooms) to the contractors, who then build the house according to their interpretation of these requirements. The requirements from the customer have a direct counterpart in the computer software, where we call such requirements a *requirements specification* or just specification for short. The specification may be more or less precise and state what is required of the computer program. This is illustrated in Fig. 1.1(a). Here we see the customer (upper left corner) write a specification (lower left corner), which is then interpreted by the contractor (middle), who builds a house, (right), based on his interpretation of the specification. As we can see, the specification is ambiguous (what is "red" and how should the $180\,m^2$ be distributed?) and incomplete (what should the roof look like?), which leads to multiple possible implementations. If the construction of the building has been outsourced to multiple contractors, they would probably interpret the specification in different ways, which could lead to houses that could not even be assembled in the end (e.g., the roof of the interpretation (I) is too small for (II)). The same problems arise when dealing with software. Ambiguities lead to software that does not do what the customer intended and different interpretations by different software companies or programmers lead to components that cannot inter-operate. In Fig. 1.1(b), we see that the customer thinks of a protocol which allows two computers to communicate in both directions over a wireless link, but the specification is ambiguous (what is "nodes") and incomplete (it does not state that communication should be bi-directional and over a wireless link), so the programmer, corresponding to the contractor in the house example, may implement the protocol under the assumption that two stationary computers communicate over a wired link (I) or as a media-server communicating uni-directionally with a computer over a satellite link (II).

In the real world, a contractor would not work directly from the specification provided by the customer. Instead, an architect would try to interpret the requirements from the customer and create a precise and complete representation in the form of blueprints. This situation is depicted in Fig. 1.2. Now the contractor is certain of how the house should be constructed, and multi-

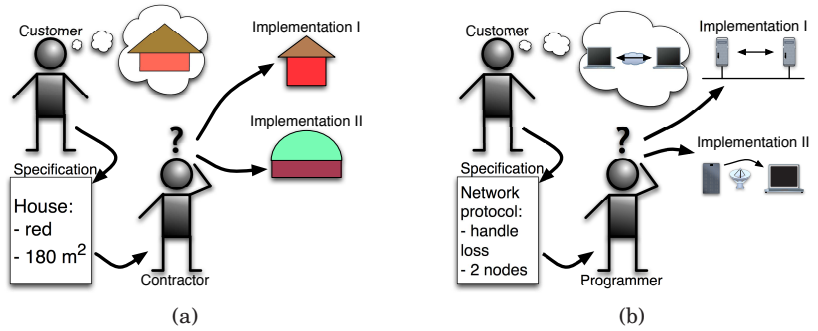


Figure 1.1: Construction directly from the specification.

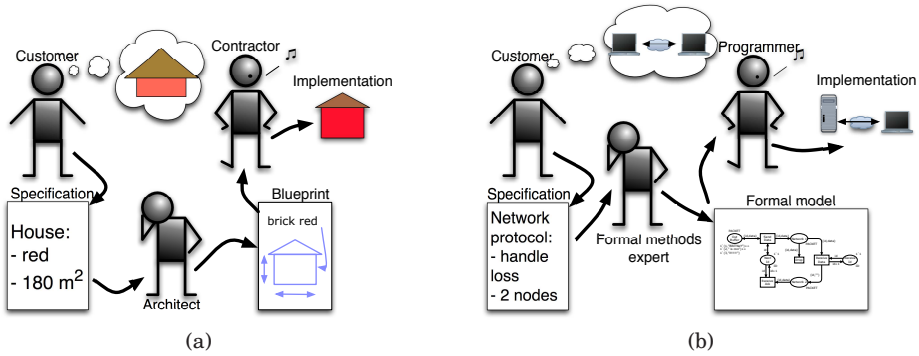


Figure 1.2: Construction using a formal model.

ple contractors can be hired to make the different parts of the house without problems. This approach is better than the previous one, as we have obtained a precise and complete description of the house we want to construct. We can use this approach for software construction as well. Here we call the blueprints a *formal model* of the program. We require that the formal model is constructed in a formal language with a formal semantics, such as coloured Petri nets [91], state charts [65], message sequence charts [67] (a variant of which is known as UML [131] sequence diagrams), Petri nets [138], CCS [123], PROMELA [77, 154], ambient calculus [17], or π-calculus [124]. We will not allow a formal model to be specified as a natural language specification or using semi-formal notations such as UML [131]. The formal model is constructed by a formal methods expert.

While the method in Fig. 1.2 ensures that we get a precise and complete formal description of what to construct, it does not ensure that the formal model corresponds to the customer's intentions. We can see in Fig. 1.2 that the house/network protocol the customer is thinking of is different from the one constructed by the contractor/programmer. This is because the original natural-language specification was not accurate or because the architect misunderstood it. We would therefore like the customer to validate that the formal description (the blueprints or the formal model) of the system corresponds to his intentions. Alas, the customer may not understand the blueprints or formal model. This may be stretching the parallel a little, as most people have some understanding of how to read blueprints, but it may not be easy to understand how the living room will look in afternoon sunlight from a set of blueprints. In any case, the customer is seldom able to read a formal model of a software

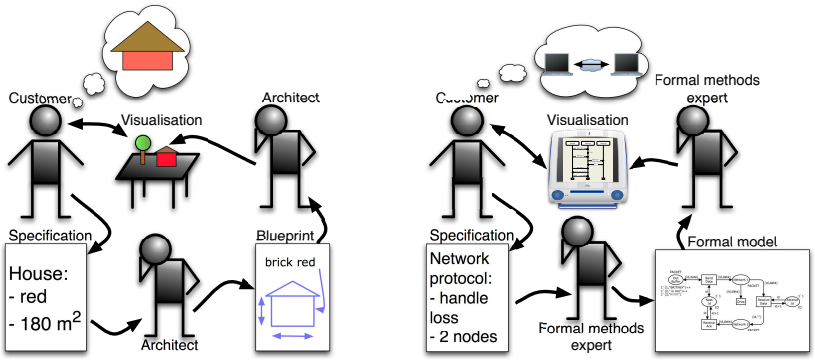


Figure 1.3: Using visualisation to ensure the formal model corresponds to the customers intentions.

system, and this is the scenario we are really interested in.

To make the customer understand the model of the system, we assume that the architect/formal methods expert or somebody else, who understands the formal model, creates a visualisation of the model. In the house example, a three dimensional physical model of the house may be constructed. It is placed in a physical model of the surroundings, which allows the customer to look at the model and see if it fits with his ideas. He may even experiment with it, e.g., by moving a lamp around it simulating how the sun looks at different times of the day. This visualisation may cause the customer to improve the specification by making requirements more precise and by specifying things missing in the original specification. In the physical world, the blueprints would then be updated and a new visualisation would be created. This is shown in Fig. 1.3(a). In the software world, we would create a visualisation of the software we are about to write, corresponding, e.g., to a prototype [44], and let the user experiment with the prototype, thereby improving the specification. Often the prototype runs as a computer program, as shown in Fig. 1.3(b).

By the method in Fig. 1.3, we can construct a complete formal model of the system we want to construct. The idea is to use this formal specification to check properties of the system we want to eventually construct. In the physical world we may want to check that the house abides by the legislation (e.g. it may be illegal to construct red houses in a certain neighbourhood), and that it is physically possible to construct the house (e.g. that the roof is not too heavy). This can be done as outlined in Fig. 1.4. Here we assume a blueprint or a formal model—which may or may not correspond to some real system—and some requirements. In the physical world an engineer would look at the requirements and the blueprints together, and either arrive at the conclusion that the blueprints satisfies the requirements, or find some errors, which are then fixed in the blueprints, e.g., by requiring that the walls of the house is made of more solid material. In the software world, we assume that the formal model and the requirements are on some form we can use as input to a verifier. The verifier can give two answers, either the requirements hold or they do not. If the requirements hold, we are satisfied with an “Ok” from the verifier, whereas we would like an error report if the requirements are not satisfied by the model. We can use this error report as input for further refinement of the model and the requirements. Sometimes the error is really an error in the model, either because we have incorrectly modelled the requirements (for

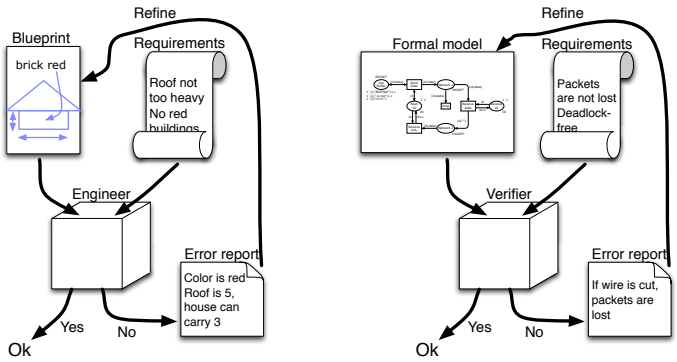


Figure 1.4: Verification of formal models.

example if the requirements specifies sufficiently strong walls but the model incorrectly specifies the strength as too low) or because the requirements are incorrect (for example if the requirements specify that walls should be too thin, so they are not able to carry the roof). In that case, we need to change the model and/or the requirements. Sometimes the error is an erroneous requirement, e.g., a too strict requirement (for example houses may be allowed to be red, just not crimson). Whenever we fix an error in the formal model, we also fix an error in the product, assuming that the product is constructed exactly as described by the formal model.

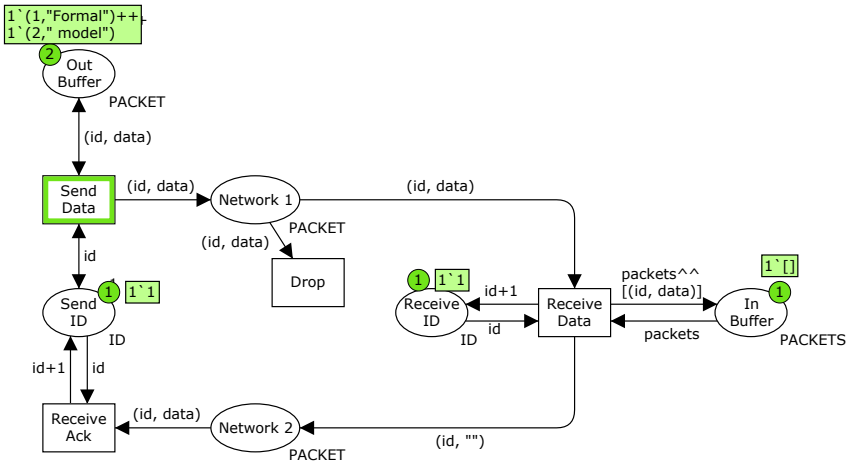
1.2 Behavioural Models of Concurrent Systems

Until now, we have talked about formal models of computer systems without defining what we mean by that. In the physical world a model of the product, e.g., a house, is an abstract representation of the product, and we want a formal model of a computer system to also be an abstract representation of the product, i.e., the implementation. The advantage of a more abstract representation is manifold. Firstly, it is often cheaper to construct an abstract model, as we do not have to deal with a lot of details, just like it is easier to draw a straight line and say it represents a wall of concrete than actually building the wall. Secondly, a more abstract representation usually has simpler behaviour, which makes analysis tractable for more complex systems.

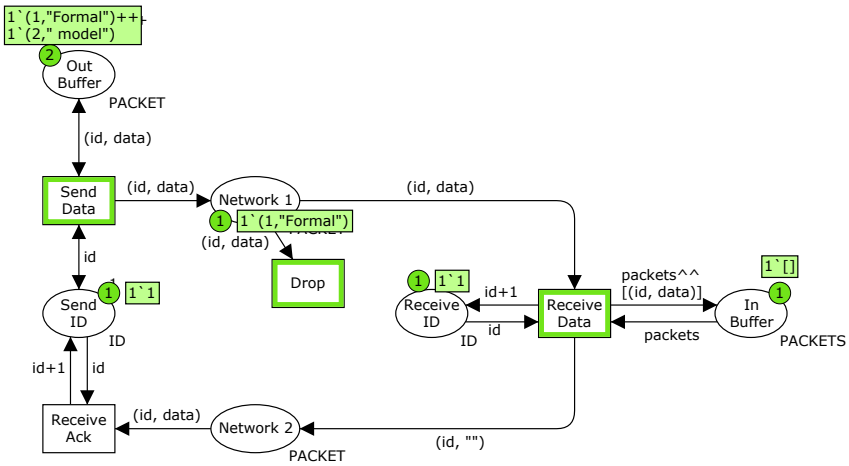
In the rest of this thesis we solely focus on models of the behaviour of concurrent systems [138], i.e., systems where computation is performed in multiple components or threads. Examples span from multi-threaded applications running on a single computer, such as a word processor which is able to continue working while sending a document to the printer, to complex distributed algorithms, such as network protocols which requires the cooperation between multiple computers connected via a network to provide a service, such as transmitting packets safely over a faulty network. We consider concurrent systems rather than single-threaded programs as the behaviour of concurrent systems is much more complex. Correct behaviour of single-threaded systems can usually be verified using, e.g., unit-testing [5], whereas concurrent systems not only depend on the input to the program, but also on the timing of each component relative to the other components, which makes is difficult to write tests that, in a reproducible way, exercise all possible interleavings of the components in question.

So, what is an abstract representation of a concurrent system? Suppose we are to create a network protocol for transmitting packets over an unreliable network. If we were to actually implement the protocol, we would need to worry about receiving actual data from the network, which is operating system-specific, we would need to decode binary data in order to put it into a form where we can process it, and we would need to set up equipment to actually test our implementation. These implementation details and many others like them make the implementation complex and may hide the real application logic. A model of a network protocol may disregard all of these implementation details, and can therefore focus on what we are actually interested in, namely the behaviour of the protocol. A real prototype or implementation would also suffer from the fact that some actions happen very rarely in reality, but when they happen the correctness of the system can be affected. As an example, packet losses happen rarely in real settings, but depending at which point they happen during the execution of a network protocol, they can have catastrophic consequences. A model of the system can be controlled, and we can intentionally drive the model into rarely occurring situations to observe the behaviour of the system in such situations.

To describe our models, we need a modelling language. Most of the work described in this thesis has been developed in the context of coloured Petri nets [91], but is independent of the formalism, and could have been created using many other formal modelling languages. A coloured Petri net is a labelled directed bipartite graph. In Fig. 1.5(a) we see a simple model of a network protocol, created in CPN Tools [C1, 33], a tool for modelling with coloured Petri nets. The model is the same as the one used in [T3] (except for typographical changes), which is a simplified version of a network protocol introduced in [91]. The nodes of a coloured Petri net are called places and transitions, and are drawn as ellipses and rectangles, respectively. The model in Fig. 1.5(a) has six places, Out Buffer, Send ID, Network 1, In Buffer, Receive ID, and Network 2, and four transitions, Send Data, Drop, Receive Data, and Receive Ack. Places have an associated type and can contain a multi-set of tokens of that type. For example, in Fig. 1.5(a), the place Out Buffer has type PACKET and contains two tokens. The number of tokens is written inside the circle next to the place and the values of the tokens are written inside the rectangle nearby. On the place Out Buffer, each token is a pair of a packet sequence number and the packet contents (i.e., of type PACKET). We see that a packet numbered 1 containing "Formal" is scheduled to be transmitted as is packet number 2 with data " model". A transition is enabled if there exist an assignment of values to all variables around it so that all the tokens required by arc expressions, with the proper values inserted, are available on input places (places connected to a transition via an arc from the place to the transition). As an example, packets are transmitted by the Send Data. This transition is enabled in Fig. 1.5(a) if we assign the value 1 to id and "Formal" to data. We write Send Data{id = 1, data = "Formal"} to represent the transition Send Data with this binding of its variables. In Fig. 1.5(a) we have indicated that Send Data{id = 1, data = "Formal"} is enabled by a green highlighting of the transition Send Data. If a transition is enabled it can be executed and the result is that tokens are removed from input places according to the arc expressions and new tokens are produced on output places (places connected to the transition via an arc from the transition to the place) according to the arc expressions. A double arc is just an abbreviation for an arc in each direction with the same expression. The result of executing Send Data{id = 1, data="Formal"} in Fig. 1.5(a) is shown in Fig. 1.5(b). Here a new packet is produced on the place Network 1, corresponding to transmitting a packet onto the network. The packet is not removed from



(a) Before executing Send Data{id = 1, data = "Formal"}.



(b) After executing Send Data{id = 1, data = "Formal"}.

Figure 1.5: A formal model of a simple protocol able to transmit packets over a network which may drop packets.

the Out Buffer, so it can later be retransmitted if needed. The newly produced packet can be dropped (Drop{id = 1, data = "Formal"}) or successfully received (Receive Data{id = 1, data = "Formal", packets = []}). When packets are received, the counter on Receive ID is incremented by one, the packet is saved in the In Buffer, and an acknowledgement is sent back to the sender, so it knows that the packet is successfully transmitted. Receive Ack receives such an acknowledgement, increments the counter in Send ID, so the sender can start transmitting the next packet. We see that this models a simple network protocol, which is able to transmit packets over a network that may drop packets. The details of how the network works have been abstracted away and packets are represented in an abstract way, so we do not have to perform complex translations of binary data, which would hide the application logic.

1.3 Verification of Formal Models

Now we focus on the verifier in Fig. 1.4. Considering the house example, the verifier would be an engineer using techniques based on the laws of physics to verify properties of the house. Naturally, we would like a way to do that in the software world as well.

In order to verify whether a formal model satisfies one or more properties, we use an analysis method. Basically, analysis methods fall into two categories: static analysis and dynamic analysis. Static analysis only looks at the description of the model whereas dynamic analysis also looks at the behaviour of the model. In the case of analysis of the network protocol described in the previous section, static analysis would only look at the model in Fig. 1.5, whereas dynamic analysis would also look at the behaviour of the model. In this Sect. we will first look at static analysis and then turn to dynamic analysis.

1.3.1 Static Analysis

Static analysis is well-known from compilers. Compilers perform static analysis to generate more efficient programs and to check for errors that may occur in programs. Static analysis is performed when the program is compiles, whereas dynamic analysis as described in the next section is performed on run-time. As an example, the Java language specification [62] cites that local variables should be *definitely assigned*[1] a value before they are used [62, Chap. 16]. In both methods in Fig. 1.6, we are interested in checking whether the variable `y` is definitely assigned a value before it is read. The variable is assigned a value in lines 4 and 11 and read in lines 5 and 12. The only difference between the two problems is that the assignment in line 10 is dependent on the evaluation of the boolean expression `x == x`. Thus, the method `definitelyAssigned` from Fig. 1.6 is allowed whereas `subtlyAssigned` is not, even though the condition of the `if` statement in line 10 would always be true. The property we actually want to check is that all variables have been assigned before they are read, but it is impossible to check this property, so we instead check the simpler property that all variables must definitely be assigned, thereby discarding `subtlyAssigned` even though it actually satisfies the desired property. Rice's theorem [146] states that any non-trivial property of the behaviour of programs cannot be checked automatically. Here trivial properties are properties that either hold for all programs or for no programs at all. Due to this property we have to translate properties stating something about the behaviour of programs into stronger properties stating something about the program. This is the strongest caveat of static analysis, as it is not always possible to find a stronger requirement that does not discard important programs.

Hoare logic [70] is a classical technique for more advanced static analysis. The idea of Hoare logic is to annotate each statement of a program with pre- and post-conditions. Pre-conditions of one statement must follow logically from the post-condition of the previous statement. Proof rules for each kind of statement makes is possible to prove post-conditions from pre-conditions. Hoare logic makes it possible to state and prove properties, such as correctness of algorithms. The main difficulty of using Hoare logic is that sufficiently strong pre-conditions must be chosen manually in order to prove the desired post-conditions.

[1] The specification of course defines this more precisely. The gist of the definition is that on all traces, i.e., where both the `then` and `else` cases of an `if`-statement are considered, all variables must be assigned before they are read.

```
1  public class Assignments {
2        public int definitelyAssigned(int x) {
3              int y;
4              y = x * 2;
5              return y;
6        }
7
8        public int subtlyAssigned(int x) {
9              int y;
10             if (x == x)
11                   y = x * 2;
12             return y;
13       }
14 }
```

Figure 1.6: Two Java methods. One is accepted by the compiler while the other is not. Neither contain any errors.

While Hoare logic requires ingenuity to come up with the correct pre- and post-conditions, a simpler variant, namely types, have become so common most programmers use them without ever really thinking about them. Types of variables are statements that the values assigned to a variable always belong to a certain type. Consider again the methods in Fig. 1.6. How do we know that the statement `y = x * 2` always makes sense? What if `x` contains the string value `"horse"`? `"horse" * 2` certainly makes no sense. We know that the statement always make sense, because we have declared that the parameter `x` must be of type `int`, i.e., that it must always contain integers. Thus, whenever we use `x`, we know that we use an integer, so `x * 2` is always successful, as multiplication is defined on integers. In addition to requiring properties of our parameter, we also promise that we always return an integer from both methods. We know that this is true, because the only value returned from either method is `y` (lines 5 and 12), and we have declared that `y` is of type integer, which this is checked whenever we assign values to `y`, such as in `y = x * 2`. The type system also prevents us from making errors like calling `definitelyAssigned("horse")`. In Java we must explicitly state the types of variables, but it is also possible to make a strongly typed programming language even without this requirement, such as Standard ML (SML) [159] or OCaml [108]. Instead of requiring the user to explicitly state the types of all variables, they can be automatically inferred from how the variables are used and consistency of the use is checked.

Type systems can also be used to check properties of certain modelling languages, e.g., the ambient calculus [17]. The ambient calculus consists of ambients, which are located inside each other. Ambients can move in and out of each other, dissolve neighbour ambients, and communicate with neighbour ambients. A problem of the ambient calculus is that it is possible to send both ambients and operations over channels. This means that we may arrive at a situation where we receive an ambient on a channel and try to execute it, assuming it is a operation, or we receive an operation and try to move inside it. Such nonsense uses of channels can be avoided if we are careful, but we can also devise a type system which checks that do not make such errors. In [15] Cardelli et al. devise three type systems for the ambient calculus, among those, one which checks that nonsense use of data received over channels does not happen. Another type system from [15] checks whether it is possible for am-

bients with a certain name to dissolve ambients with another name (which can be bad if, e.g., it is possible to send two packets to a remote computer and one packet contains code which is able to open the other packet to unveil a virus). The full result of [15] is a type system, which, in addition to checking he aforementioned two properties, also checks whether ambients with a certain name are able to enter ambients with another name, which can, e.g., be used to check the effectiveness of firewalls (if malicious code is able to enter through the firewall it is not effective). Ambients need not be explicitly typed, much like how values do not need to be typed in SML, but any process that can be correctly typed exhibit the desired behaviour at run-time. Type systems need to be developed and proved correct for each kind of property we would like to check, and are therefore not that applicable for proving arbitrary properties, but useful for guaranteeing absence of a certain kind of errors. Furthermore, type systems are useful when translating from one formalism to another using a translation inductive in the structure of the source formalism. By explicitly assigning types to the result of the translation, we can inductively prove absence of a certain kind of errors (the kind guaranteed by the type system) in all translated models.

Type systems are a special case of invariants. Invariants are properties that must always hold during the entire execution of a program or formal model. Type systems are invariants stating that a given variable always contains a value from a given set or that something sent over a given channel is always a channel, and Hoare logic uses invariants in loops. Of particular interest are invariants that can proven solely by looking at the program or the formal model. Coloured Petri nets also allow the specification of invariants—in fact two dual kinds of invariants: transition invariants and place invariants. Transition invariants state that the effect of executing a certain multi-set of transitions (provided there are enough tokens initially) is the same as executing no transitions at all. In Fig. 1.5(a) the effect of executing the transition Send Data{id = 1, data = "Formal"} is adding one token to Network 1 and the effect of Drop{id = 1, data = "Formal"} is the exact opposite. Thus we have a transition invariant 1‘Send Data{id = 1, data = "Formal"} ++ 1‘Drop{id = 1, data = "Formal"} (using the multi-set notation of CPN Tools). A place invariant is a set of weight-functions, which, when applied to the tokens available on all places, always yields the same value. In Fig. 1.5(a), if we map any integer to 1‘0 on Send ID and Receive ID and all multi-sets of tokens on other places into the empty multi-set, we get an invariant, as this always yields 2‘0. The weight functions should be linear in the number of tokens, and can map into any domain desired. Invariants of CP-nets are not really that useful for analysis, as they must be interpreted manually depending on the model. Furthermore, calculating invariants requires calculation of inverse functions of the functions appearing in the arc-expressions, which is not possible for CP-nets as the functions appearing in the arc-expressions can be arbitrary, making the calculation of invariants uncomputable. Work on automatically checking invariants for CP-nets has been implemented and shown to work on a small set of examples by Toksvig in [158]. Invariants are more interesting for a simpler kind of Petri nets, namely Place-transition Petri nets (PT-nets) [36]. PT-nets can be considered as a simplified version of coloured Petri nets, but actually predates coloured Petri nets. PT-nets are like coloured Petri nets except that all tokens are equal—the type of all places must be UNIT $= \{\bullet\}$ (written () for technical reasons) and all arc expressions must be integers, signifying how many tokens are moved from each place. A PT-net version of the network protocol from Fig. 1.5 is shown in Fig. 1.7. The protocol in Fig. 1.7 is only able to transmit a single packet, but otherwise behave like the coloured Petri net version. All

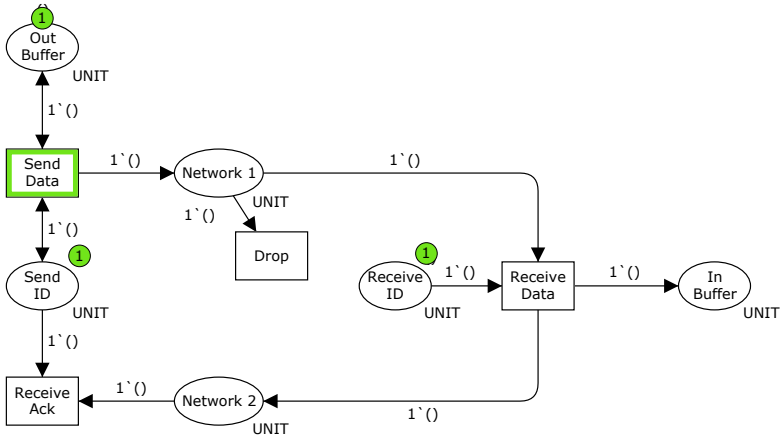


Figure 1.7: A simplified version of the network protocol from Fig. 1.7 created as a PT-net.

places now have type UNIT and the values of the tokens have been removed. As no variables exist on any arcs, we no longer have to specify the variables when discussing enabling. The transition invariant 1‘Send Data ++ 1‘Drop is preserved. The advantage of considering PT-nets when calculating invariants is that we only need to compute the inverse of linear functions on integers, which is computable.

As mentioned, due to Rice's theorem, it is not possible to answer questions about a model's behaviour (at least models described using a Turing Complete [11] modelling language such as CP-nets) or a programs execution by only looking at the model or program itself. For debugging it is not a problem that we are unable to give exact answers, as we are often satisfied with being told about potential errors or proving absence of simple errors, explaining the success of static analysis, which gives a sound answer, meaning that if the analysis states that no error exists, then no error exists (of the kind we check). Static analysis is not complete, though, so if static analysis discovers a problem, this does not necessarily mean that there really is an error. Some properties are very difficult to determine using static analysis at a level fine grained enough to be really useful, however. Such properties include whether allocated memory is always freed exactly once, and whether buffers can over- or under-flow. When we want to ensure that some run-time property holds, an approximate answer may not be enough. The idea of dynamic analysis methods is to explore the behaviour of the system during run-time.

1.3.2 Dynamic Analysis

The simplest way to check the run-time (dynamic) behaviour of a system is to test it, i.e., execute the system a number of times and manually or automatically check that the behaviour corresponds to the desired behaviour. Tests can be written manually or created automatically [45] by a computer tool. When dealing with models, we usually refer to the execution of the model as a *simulation* of the model. Tests makes it possible to unveil errors that are difficult to find with static analysis, but do not ensure absence of errors. In order to ensure absence of errors, we need to ensure that our tests cover all possible execution paths. In order to do that, we build a reachability graph (also known as a state space). A reachability graph is a directed labelled graph, where the

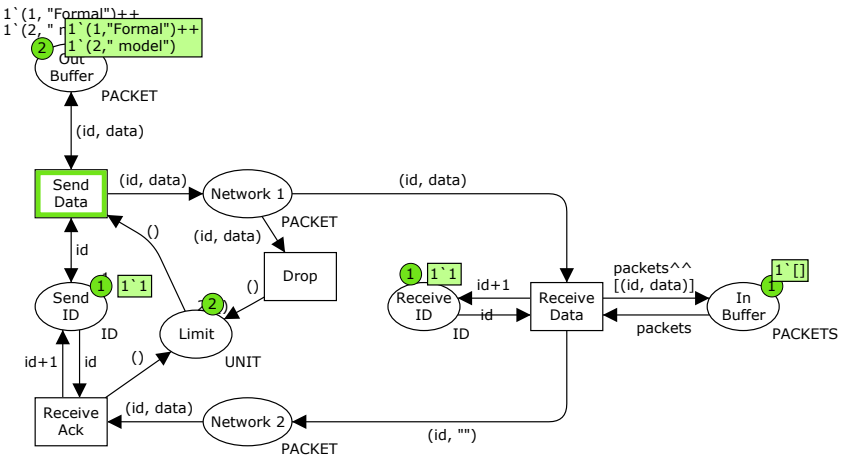


Figure 1.8: A version of the simple network protocol from Fig. 1.5 with a bound on the number of possible outstanding packets.

nodes correspond to states of the model and a labelled arc from one node to another, signify that it is possible to go from the state represented by the source to the state represented by the destination using the transition (and, in the case of CP-nets, the binding) corresponding to the label of the arc. If we have the reachability graph available, we can check all properties we can think of. The problem of this method is that it is difficult to construct the reachability graph, either because the reachability graph is very large or because it is infinite. As an example in the case of the model in Fig. 1.5, the reachability graph is infinite, as we can just keep executing the transition Send Data{id = 1, data = "Formal"}, producing an arbitrary number of tokens on Network 1, producing a new state for each number of tokens on Network 1. If we limit the number of tokens we can put on Network 1 and Network 2, the reachability graph becomes finite, however. The modified model can be seen in Fig. 1.8. The change is that we have added a place Limit, which initially contains two uncoloured tokens. Whenever we add a token to Network 1 or Network 2 we remove a token from Limit and vice versa. In that way we ensure that there is at most two tokens simultaneously on Network 1 and Network 2. Having made this change, we obtain the reachability graph in Fig. 1.9. The red node is the initial state and the green node is the only state with no successor states. The detailed specification of a state, the *state descriptor*, is written inside the node. Each state is represented by the value of the token on the Send ID place, the sequence numbers of the packets in Network 1, the number of tokens available on Limit, the sequence numbers of the packets on In Buffer, the value of the token on Receive ID, and the sequence numbers of the packets on Network 2. The transitions are represented by an abbreviated version of their name and the sequence number of the packet being processed. Using this reachability graph, we can, e.g., see that packets are always received in-order, regardless of how packets are lost. We see this by observing that on all states of Fig. 1.9 the value of the In Buffer is either "" (no tokens), "1" or "1; 2". As the tokens are shown in the order the packets with the corresponding packets have arrived, we see that we never encounter the situation "2; 1" or "2", where the packets are received out of order.

To alleviate the problem that the number of states is very large or infinite, also known as the *state explosion problem* [161], we can do several different things. One idea is to store the reachability graph in a more efficient way or to

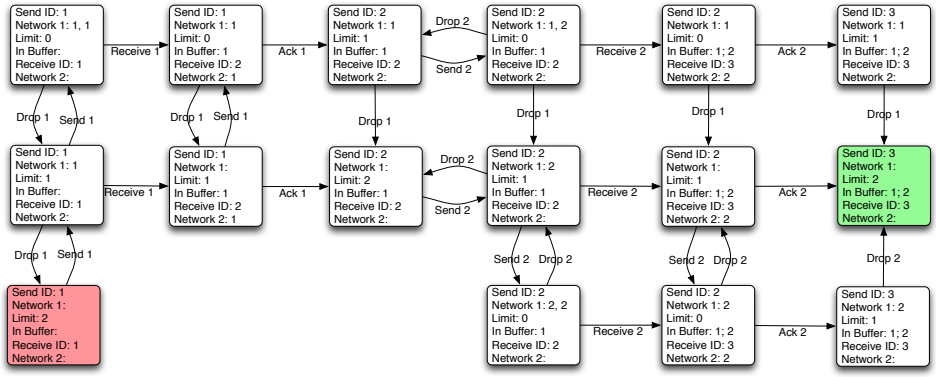


Figure 1.9: The reachability graph of the simple network protocol in Fig. 1.8.

only store enough information that we are later able to reconstruct the reachability graph. Construction of the reachability graph can be done in two ways, either explicitly or symbolically. Explicit reachability graph analysis explicitly store the reachability graph in memory, whereas symbolical reachability graph analysis only has an implicit representation, e.g., by representing all states as a logical formula satisfied by exactly the reachable states. For explicit reachability graph analysis, we often use a reduction technique in order to only require as much internal memory as is available. Examples of reduction techniques are the sweep-line method [T1, 25, 104], which uses a notion of progress in the model to delete states that cannot be reached again, hash compaction [155, 172], which does not store an actual representation of the state descriptors, but only a hash value calculated from the state descriptor, called a *compressed state descriptor*, and the ComBack method [T2], which is an extension of hash compaction solving the problem of hash collisions, which arise when two state descriptors have the same hash value, meaning only successors of one of the states is considered; by maintaining a spanning tree of the reachability graph, it is possible to reconstruct the full state descriptors and resolve hash collisions. Symbolic reachability graph analysis typically use, e.g., binary decision diagrams [12] or multi-valued decision diagrams [96] to store states efficiently.

Another approach is to only guarantee properties on traces of some finite length. This is known as bounded model checking [8], and relies on tools that are able to solve the SAT problem [148] for propositional logic, i.e., whether there exists an assignment to all propositional variables of a given propositional formula, such that the formula evaluates to true. Bounded model checking is also an instance of symbolic model checking, where states are represented using boolean formulae.

Another idea is to create a coverability graph [52, 97] instead of a reachability graph. This method is specific to Petri nets, but the coverability graph is always finite and allows us to check certain interesting properties, e.g., to find maximum number of tokens on all places. We get into more detail about reduction techniques in Chapter 2.

1.4 Behavioural Visualisation of Formal Models

When we have created a formal model of a concurrent system like the network protocol in Fig. 1.5, we would like to make sure that the constructed model

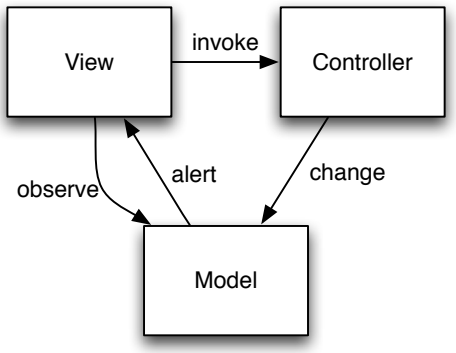


Figure 1.10: The Model-View-Controller design pattern.

corresponds to the intended system using the approach in Fig. 1.3. To introduce visualisations of formal models, we will first introduce the Model-View-Controller (MVC) [100] design pattern [54], which is the foundation for many approaches to visualisation of formal models.

1.4.1 The Model-View-Controller Design Pattern

A design pattern [54] is a recipe for how to do a certain task in a programming language. The Model-View-Controller (MVC) [100] design pattern is a recipe on how to create graphical user interfaces that are able to manipulate a data structure within the computer. The data structure may represent, e.g., a text document or the organisation of a company. When using the MVC design pattern, the data structure we wish to manipulate is called the model (not to be confused with formal models as discussed previously). The user interface the user see is called the view, and the code able to cause changes to the model is called the controller. In Fig. 1.10 we see how the three parts of the system interact. When a user wishes to create a change in the model, an action in the user interface, i.e., the view, is triggered. An example of this is when a button is clicked or an item is selected from a menu. This causes the view to invoke the corresponding function in the controller. The controller then changes the model accordingly, e.g., removes a line of text or promotes a salesman to manager. When the model is changed, it alerts the view, which observes the model and updates itself accordingly. This gives the user the impression that the desired update was performed in the user interface and that work is done on the graphical view of the model rather than on the underlying model itself.

One important consequence of using the MVC design pattern is that it is possible to have more than one view for each model. When a change is made in one view, all other views are updated as well. This happens because the model alerts all views whenever a change occurs. As an example, consider the interaction depicted in Fig. 1.11. Here two views, View 1 and View 2, are associated with a single model. The figure shows that a user initiates an action in View 1. This causes the view to invoke the corresponding code in the controller. The controller then changes the model accordingly. The model then alerts both views, which causes them to observe the model anew. Depending on the implementation, the model may alert all views before any of them update themselves according to the model, or each alert may be immediately followed by an update. In Fig. 1.11 we assume that all alerts happen before any observations. The views can show the same or they can show different aspects of the model.

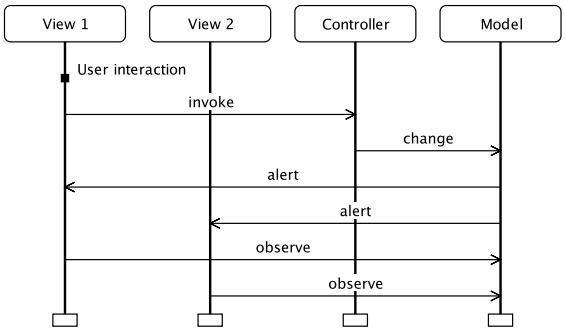


Figure 1.11: How two views associated with the same model are updated.

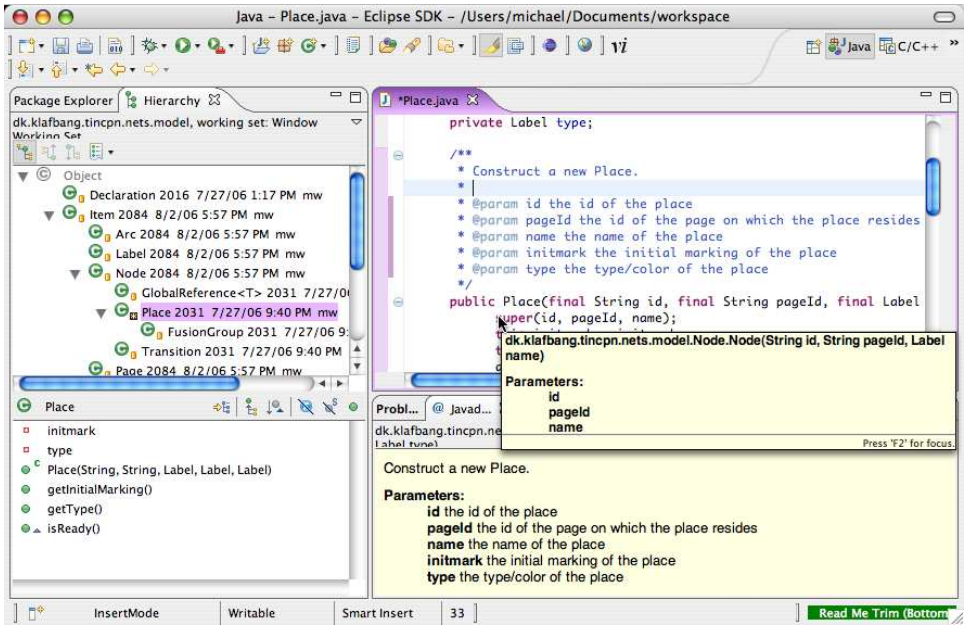


Figure 1.12: Screen-shot from the Eclipse Java editor.

Consider the screen-shot from Eclipse [41] in Fig. 1.12. Eclipse is a tool for editing Java programs. Here we see five different views on the class Place from a coloured Petri nets editor. At the upper right we see the actual code of the class, and at the lower right we only see the embedded documentation of the constructor of the class. At the lower left we see an overview of the class, and at the upper left we see a partial overview of the class hierarchy including the class Place. Finally, we can see a tool-tip near the mouse in the middle of the image, which shows an abbreviated version of the documentation for the item under the mouse. All of these views are different views of the same model and we see that they show different details about the model. Some show (nearly) all details and some show very limited details about the model, but whenever a change is made to one of the views, e.g., if the class is modified in the upper right window, all the views are updated automatically.

We now let the formal model be the model of the MVC design pattern and we let the visualisation be the view. The controller is usually the tool used to simulate the formal model, but may also be integrated with the visualisation.

1.4.2 Behavioural Visualisation of Formal Models Using the Model-View-Controller Design Pattern

If the concurrent system we wish to develop and thus model and visualise is a simple form-filling application, such as a business intelligence or inventory application, a visualisation can quite easily be constructed using the idea of a prototype [44]. A prototype is a simple implementation of a program in which only limited functionality has been implemented, but otherwise the prototype looks and behaves as the real implementation. In MVC terms we implement only the view and very simplistic models and controllers. Prototypes are valuable as a tool for testing a user-interface before a costly construction of the real product. The idea is that it is very easy and cheap to create a reasonably professional-looking user interface using a GUI-builder, such as Borland JBuilder [90] or Microsoft Visual Studio [167]. Normally, we would then extend the purely visual prototype with simple code that make the prototype act as expected of the real program. If we, instead of a simplistic implementation of the model and controller, use a formal model in place of the model and the simulation tool as controller, we get a product whose behaviour is defined by a formal model. As the GUI is a view of the formal model, it is possible to see the state of the formal model. Whenever the formal model's state is updated, the GUI is updated accordingly. By letting actions performed in the GUI correspond to actions in the formal model, it is also possible to stimulate the formal model, and it is thus possible to see and stimulate the execution of the formal model using a standard GUI.

Some times the model is not modelling a simple form-filling application, however. As an example, the network protocol from Sect. 1.2 is a more complex system. It is not obvious how we should create a user-interface that allows us to observe the behaviour of the system as a network protocol would not have a graphical user-interface except for configuration purposes. We could, however, create a visualisation rooted in the network diagrams used to diagram the layout of a large network, where all machines are drawn as icons and packets as coloured dots like the one in Fig. 1.13. The figure shows the sender to the left and the receiver to the right. The cloud represents the network. The coloured dots represent packets; green packets contain data en route from the sender to the receiver while red dots correspond to acknowledgements en route in the other direction. The number in the dots shows the sequence number of the packet. Below the sender and receiver, we see counters, representing the counters on Send ID respectively Receive ID. Currently both of these are 1. We may be able to transmit packets by clicking on the sender. The graphics in Fig. 1.13 is updated while packets are transmitted, e.g., to show whether packets are dropped or successfully received as well as to show the current values of the counters. If we implement code which is able to show and maintain a visualisation like the one in Fig. 1.13, we can use it as view, the formal model as model and the simulation tool as controller as in the case of the GUI application. This is an example of a domain specific visualisation, as we have used a visualisation that is likely to be familiar to the domain expert.

In this manner we can implement a model-based prototype, which has several advantages over a normal prototype or an implementation. As an example, it is possible to abstract away certain implementation details. In the case of the network protocol, we are able to ignore any operating system-specific network access and encoding/decoding of binary data. Compared to creating a prototype written entirely in, e.g., Java we also obtain a formal specification of the system we wish to implement without representing the dynamics of the protocol twice, once in the prototype and once in the formal model. The formal

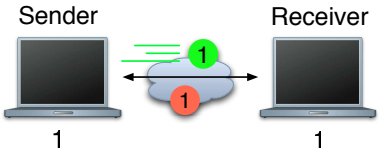


Figure 1.13: Visualisation of a simple network protocol.

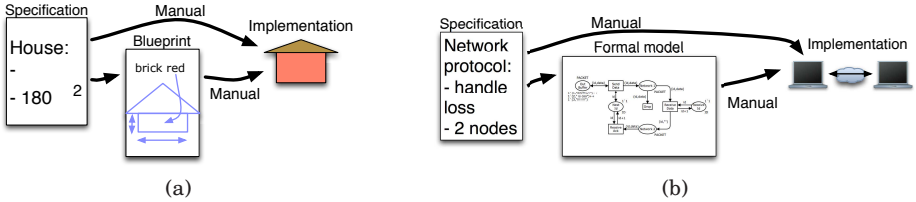


Figure 1.14: Manual construction.

model can be used for analysis or as basis for an implementation as discussed in the next section. 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 formal modelling language.

1.5 Relationship between Formal Model and Implementation

Once we have constructed a formal model, we validate that it reflects the system we want to construct using the approach shown in Fig. 1.3 and described in Sect. 1.4. Then, we verify that the model satisfies the requirements we may have using the method shown in Fig. 1.4 and described in Sect. 1.3. The next step is then to actually implement the system. In this section we look at four ways to arrive at an implementation based on a formal model.

The most straightforward way to obtain an implementation corresponding to a formal model is to look at the formal model and the specification and manually create the implementation, relying on experience to make a reasonable translation. This approach is shown in Fig. 1.14(b). This approach corresponds to how we would build a house from the architect's drawings (Fig. 1.14(a)). Advantages of this approach are that it is light-weight, easy to understand, and easy to start using: it is easy for somebody who understands the formalism used to describe the formal model to create the implementation. This approach is also the one seeing the widest use, and has been described under some form as the waterfall model [147], and the idea also underlies the widely used Capability Maturity Model (CMMI) [30]. The major disadvantages are that the manual step is prone to human errors, and a lot of difficult decisions are hidden in the art of the manual translation. This approach can be used with any reasonable formalism and any tool, as the modelling phase is only present to clarify the specification.

An obvious way to make the approach less prone to human errors is to eliminate the manual step from the formal model to the implementation and let a computer create the final system from the formal model. This can be seen Fig. 1.15. In the real world this corresponds to building a machine that builds houses from the architect's drawings with no human intervention. This ap-

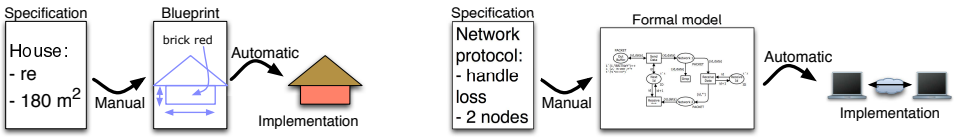


Figure 1.15: Synthesis.

proach actually solves both of the problems with the manual approach. As the step from model to the implementation is automatic, it is not possible for humans to introduce errors in this step. Also, as we have to construct the machine constructing the implementation from the model, we cannot hide difficult decisions. We have to find a solution to all difficult problems or the machine will not work. This corresponds to how high-level languages translate "easy-to-understand" programs written in high-level languages as Java or C# into lower level byte-code, which can be executed by (virtual) machines. The problem is that it is not obvious how this can be done without introducing limitations to what kinds of systems can be built or making the modelling language very complex. Currently, successful attempts at this method either restrict themselves to a certain domain, e.g. workflow modelling [164] as implemented by Machado et al. in [114], or they limit themselves to creating skeleton programs only, i.e., programs where only the main structure is automatically derived, and all the details have to be filled in by humans as done by, e.g., Hauser and Koehler in [68]. Thus the step from model to implementation is semi-automatic only.

Another approach, which is not feasible in the physical world, is to manually construct the implementation and automatically derive the model from the implementation, as shown in Fig. 1.16. Should we try finding a parallel in the physical world, we can compare this method to creating a blueprint of a house after it has been built by measuring the size of all rooms. This method is intended to find errors in the implementation and builds on the fact that comprehensive testing of the actual implementation is typically computable infeasible, whereas testing an abstraction, a model, may be feasible. The idea is to automatically derive a model from the implementation and verify formal requirements on the model. If a requirement is not satisfied by the model, we retry the exact same test in the implementation to verify if the error is reproducible there. If it is, we must fix the implementation, derive a new model and re-run the test. If the error is not reproducible in the implementation, we must refine the model until it is no longer possible to reproduce the error in the model. The major advantage of this method is that it really does find errors, as can be illustrated by two example implementations: One implementation is Holzmann and Smith's FeaVer [51, 79], where a human assists the computer in deriving the formal model from programs written in the C programming language [98]. Refinement is done manually as well, if required. FeaVer has been used to verify Lucent's PathStar^{TM} access server for telephony [80]. A more recent method is to fully automatically derive the model from the program and automatically refine the model based on automatic testing against the implementation, as implemented in Microsoft's SLAM [4, 152] for testing device drivers. As device drivers run in a privileged mode in the operating system they have the ability to crash the entire computer when failing, so correct operation is important. As can be seen from the examples, some of the major players in the computer industry are interested in this approach as it is very well-suited and efficient for finding errors in programs. The approach is fairly easy to use, but it is still too time-consuming and costly to use this

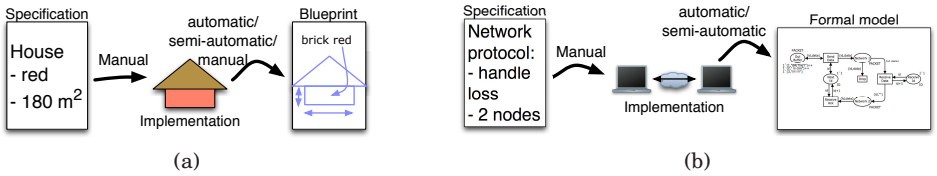


Figure 1.16: Testing using automatically generated formal models.

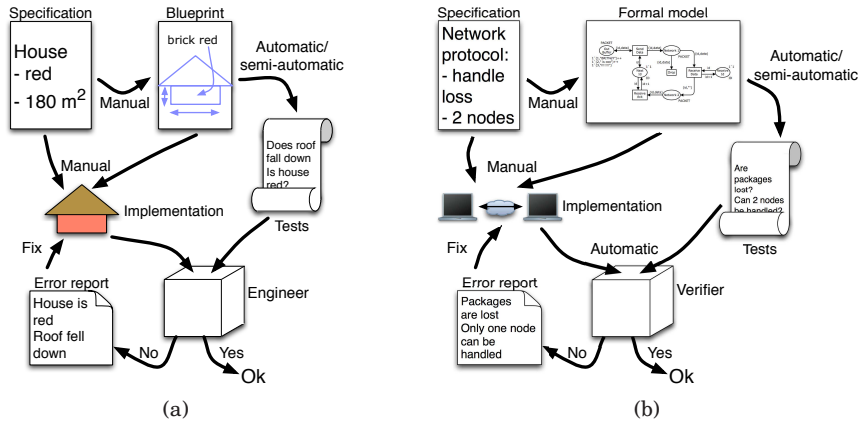


Figure 1.17: Using formal models to generate tests of the implementation.

method for non-critical systems. The major disadvantage of this approach is that it solely focuses on finding errors after the implementation has been created, which may be much more expensive than finding and fixing the error before implementation is even started. Another disadvantage is that the automatically derived abstract model may be less efficient than a humanly derived abstraction, making it infeasible to analyse.

The final approach to correct systems we will consider in this thesis is a combination of all the previous methods. This approach is outlined in Fig. 1.17. The idea is that we manually construct a model from the specification (and ensure that is correspond to the customer's idea of the system using visualisations as in Fig. 1.3). We can then verify that the model satisfies the requirements, using the verification approach in Fig. 1.4. After all requirements have been successfully verified, we construct the implementation from the specification and the model (manually as in Fig. 1.14 or automatically/semi-automatically as in Fig. 1.15). Now we automatically or semi-automatically derive tests from the model. The tests are run on the implementation and errors in the implementation revealed during this are then fixed. This approach has a lot of the advantages over the previous methods: it is possible to find errors early in the construction, it is fairly easy to get started, and the tests of the implementation ensures that the number of human mistakes introduced by going from the model to the implementation is minimised. We can check the behaviour of the implementation against the model by simply executing the two in parallel and check whether it is possible for the implementation to do something which is not allowed by the model. This has, e.g., been done by Larsen et al. in UPPAAL-TRON [107]. Of course the quality of the results is dependent on the quality of the model, as errors in the implementation also present in the model will not be reported, so validation of the model is still important prior to testing.

The most suitable approach depends on the situation. If all we want to do

is to find errors in programs which have been written, e.g., ten years ago, we should use the approach in Fig. 1.16. If we are in a situation where an implementation can automatically be synthesised from the model, we should use the method in Fig. 1.15 and skip or significantly shorten the testing phase. If resources are limited or the importance of the implementation is very limited, we may use the approach in Fig. 1.14 (maybe even skip the modelling phase and go directly from specification to the implementation) and save resources for more critical projects. If none of the other apply, the method in Fig. 1.17 may be applicable, as it makes fewer assumptions of the system to implement.

1.6 Reading Guide

This thesis in structured as follows: In Chapter 2 we consider analysis of formal models using the reachability graph method. The contribution in this area consists of two new reduction techniques. In Chapter 3 we look at different ways and tools to visualise the behaviour of a formal model. This chapter can be read independently of Chapter 2. The contribution in this area consists of the development of a tool, the BRITNeY Suite, facilitating visualisation of formal models as well as the development of a general framework for tying visualisations to formal models, giving visualisations a formal semantics, which makes it possible to visualise error reports from reachability graph analysis. In Chapter 4 we summarise the first part of this thesis. Part II of the thesis (Chapters 5—8), contains papers by the author of this thesis within the fields of reachability graph analysis (Chapters 5 and 6) and behavioural visualisation of formal models (Chapters 7—9).

To make it easier to distinguish papers that are part of this thesis and papers co-authored by the author of this thesis from papers authored by others, references to papers that are part of this thesis are prefixed with a T (for thesis), as in [T2], references to papers that are not part of this thesis but co-authored by the author of this thesis are prefixed with a C (for co-authored), like [C5], whereas other papers have no prefix, e.g., [91].

1.6.1 Brief Summary of Papers

Here we give a very brief summary of the papers in Part II of this thesis. For more extensive summaries and discussion of the papers, readers should turn to Chapters 2 and 3.

Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method

[T1] T. Mailund and M. Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 177–191. Springer-Verlag, 2004.

This paper extends the sweep-line method [25, 104] to allow checking properties that are more complex than invariants by generating a near-optimal representation of a reachability graphs using the sweep-line method. The idea is to represent states using a number and only maintain a mapping from state numbers to state descriptors for a limited set of states, namely the states in front of a sweep-line, which tries to separate states that still needs exploring from states that have already been explored and will not be encountered again. The

method is demonstrated to use significantly less memory on examples where there is a clear notion of progress, i.e., where there are few transitions leading to states that have already been explored. In addition, the method performs reasonable for examples without no clear notion of progress.

The ComBack Method—Extending Hash Compaction with Backtracking

[T2] M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *Proc. of ATPN'07*, volume 4546 of *LNCS*, pages 446–464. Springer-Verlag, 2007.

The idea of the ComBack method is to augment the hash compaction reduction technique [155, 172] by maintaining a spanning tree from the initial state to each encountered state. Hash compaction creates a compressed state descriptor from the original state descriptor using a hash function. Hash collisions, i.e., when two different state descriptors have the same compressed state descriptor, makes this method incomplete. Using the ComBack method we can use the spanning tree to translate each compressed state descriptor to all corresponding state descriptors, making it possible to discover hash collisions on-the-fly. The method is demonstrated to use around 25% of the memory required to store the reachability graph at the cost of using $100\% - 1000\%$ of the time.

The BRITNeY Suite Animation Tool

[T3] M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.

This paper describes the BRITNeY Suite visualisation tool, which makes it possible to visualise the execution of formal models. The tool is able to interact automatically with CPN Tools [C1, 33], a tool for editing and simulating coloured Petri nets. The tool allows the use of extension plug-ins, which makes it easy to extend the tool with new kinds of visualisations, but the tool also comes pre-packaged with around 20 plug-ins, making it easy to get started. The usefulness of the tool is demonstrated using two industrial case-studies.

Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks

[T4] L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.

This paper describes an industrial case study where coloured Petri nets have been used to create a prototype of a network protocol. The prototype uses the BRITNeY Suite for visualisation of the behaviour of the model (much like the approach in Fig. 1.3). The prototype has been used for discussing the model during model- and protocol-design as well as for demonstration for management with little knowledge of formal models. The paper argues that a model-based prototype can be much more efficient than a physical prototype, as we are able to abstract implementation details away and we do not have to worry about real hardware, which makes it easier to control scenarios and easier to scale the prototype.

A Game-theoretic Approach to Behavioural Visualisation

[T5] M. Westergaard. A Game-theoretic Approach to Behavioural Visualisation. Submitted, 2007.

A lot of different tools supporting visualisation of the behaviour of formal models exist, but they are typically designed in an ad-hoc manner, which often means that the semantics of the visualisation is not well-defined. Furthermore, the tools usually mainly allow simple inspection of the formal model during execution, or require that the user spends a lot of time tying the visualisation to the model. This paper regards visualisations as games, i.e., labelled transition systems where the transitions are separated into controllable and uncontrollable transitions. Visualisations are synchronised with models, whose semantical domain also is games, such that uncontrollable transitions of the model is synchronised with controllable transitions of the visualisation and vice versa. The paper gives two example visualisations and provide an application, namely visualisation of error reports of reachability graph analysis.

Chapter 2

Behavioural Verification by Means of Reachability Graphs

This chapter considers the implementation of the verifier box from Fig. 1.4. The job of the verifier is to check whether a model, denoted by $\mathcal{M}$, satisfies a given property, denoted φ. If $\mathcal{M}$ satisfies φ, we say that $\mathcal{M}$ is a model of φ, and we write $\mathcal{M} \models \varphi$. The task of checking whether $\mathcal{M} \models \varphi$ is called *model checking*.

In this chapter we will introduce the basic idea behind reachability graph analysis (also known as state space analysis) and a number of reduction techniques, i.e., variations of the basic reachability graph algorithm that make analysis possible for larger systems of certain classes of models (which class depends on the reduction technique). We start by introducing the basic algorithm for reachability graph construction in Sect. 2.1. We then turn to describing reduction techniques in general in Sect. 2.2, and the sweep-line method [25, 104] in Sect. 2.2.1 and the hash compaction reduction technique [155, 172] in Sect. 2.2.2. We give a summary of the papers [T1] and [T2] co-authored by the author of this thesis in Sects. 2.3 and 2.4. Full versions of the papers [T1] and [T2] can be found in Chapters 5 and 6, respectively. The paper [T1] extends the sweep-line method to allow checking more complex properties and [T2] makes the incomplete hash compaction reduction technique complete. We sum up the chapter by discussing the contribution of the papers [T1] and [T2] and provide directions for future work.

2.1 Basic Reachability Graph Analysis

To make our discussion of behavioural verification independent of the concrete modelling formalism, we will use an abstract definition of the behaviour of a formal model, namely a labelled transition system. A labelled transition system captures the intuition that a formal model starts in a certain state and progresses according to a transition relation:

Definition 2.1 (Labelled Transition System) *A **labelled transition system (LTS)** is a tuple, $\mathcal{LTS} = (S, T, \Delta, s_I)$, where*

- *$S \neq \emptyset$ is a set of **states**,*
- *T is a set of **transitions**,*
- *$\Delta \subseteq S \times T \times S$ is the **transition relation** indicating **successor states**,*
- *$s_I \in S$ is the **initial state**.*

Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. An *occurrence sequence* is an alternating sequence of states s_i and transitions t_i written $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_n \xrightarrow{t_n} s_{n+1}$ and satisfying $s_i \xrightarrow{t_i} s_{i+1}$ for $1 \leq i \leq n$. We use $\rightarrow^*$ to denote the transitive and reflexive closure of Δ, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_n \xrightarrow{t_n} s_{n+1}$, $n \geq 1$, with $s = s_1$ and $s' = s_{n+1}$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$, and $\mathsf{reach}(s) = \{\, s' \in S \mid s \rightarrow^* s' \,\}$ denotes the set of states reachable from s.

We will often like to verify some *invariant property*, $\mathcal{I} : S \rightarrow \{\mathsf{tt}, \mathsf{ff}\}$, of all states reachable from the initial state, i.e., check whether $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$ holds. The most naive way to do that is by checking if it holds for the initial state. If it does not, we know the property does not hold for all states. If the property does hold for the initial state, we recursively check the property for all successor states. It is evident that this algorithm does not terminate if the invariant holds and we can reach an infinite number of states, i.e., if $|\mathsf{reach}(s_I)| = \infty$ and $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$. Even if the number of reachable states is finite, the algorithm will not terminate if the invariant holds and it is possible to reach some state state from itself by a non-empty transition sequence, i.e., when $|\mathsf{reach}(s_I)| < \infty$ but $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$ and $\exists s_b, s \in \mathsf{reach}(s_I).s_b \rightarrow^* s' \rightarrow s_b$, as the recursive check of successor states will eventually encounter the state s_b, and, as $s_b \rightarrow^* s_b$, loop when trying the validate the invariant for s_b.

In order to overcome this problem, we build a reachability graph. A reachability graph is a directed labelled graph, where the nodes correspond to states of the model and a labelled arc from one node to another, signify that it is possible to go from the first state to the second using the transition corresponding to the label. Formally, the reachability graph is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges. An edge (s, t, s') has s as source and s' as destination and the label is t. The reachability graph can be constructed using Algorithm 1, which makes the recursion stack explicit as the data-structure W. The intuition is that W contains states for which we have not yet calculated successor states whereas V and E contains the nodes, respectively edges, of the reachability graph for the states for which we have already calculated successor states. This algorithm not only terminates as long as $|\mathsf{reach}(s_I)| < \infty$, it is also more efficient than the previous algorithm, as successors are only calculated once for each state. Using the reachability graph, we can traverse all states of V and check the invariant property $\mathcal{I}$, even if S is infinite as long as V is finite. Algorithm 1 terminates iff $|\mathsf{reach}(s_I)| < \infty$ and $\forall s \in \mathsf{reach}(s_I).|\{(s, t, s') \mid (s, t, s') \in \Delta\}| < \infty$. This is a dynamic property, however, and can only be decided by generating the reachability graph (or something equivalent). To obtain a syntactic way to decide if the reachability graph is finite, we observe that $\mathsf{reach}(s_I) \subseteq S$ and $\forall s \in S.\{(s, t, s') \mid (s, t, s') \in \Delta\} \subseteq \Delta \subseteq S \times T \times S$, so it is a sufficient but not necessary condition that S and T are finite for Algorithm 1 to terminate. If a Place-transition Petri net is *bounded*, i.e., if the number of tokens on all places in all reachable states is less than some constant, the set of possible states, S, is finite (or can be picked to be finite). If we furthermore assume that the PT-net only has a finite number of transitions, Algorithm 1 always terminates. Some PT-net models are bounded by design. As an example, 1-safe PT-nets only allow transitions to be executed if it does not lead to more than one token on any place. Initially all places contain at most one token, so the number of tokens never exceed 1. Some PT-net models can be shown to be bounded, e.g., using place invariants or coverability graphs as described later. We can imple-

ment this algorithm by representing V and E as hash tables and W using, e.g., a queue or a stack.

Algorithm 1 Basic reachability graph algorithm.

Require: $\mathcal{LTS} = (S, T, \Delta, s_I)$ a labelled transition system
Ensure: (V, E) the corresponding reachability graph

```
1:  V := {s_I}
2:  W := {s_I}
3:  E := ∅
4:
5:  while W ≠ ∅ do
6:     Select an s ∈ W
7:     W := W \ {s}
8:     for all t, s' such that s --t--> s' do
9:        E := E ∪ {(s, t, s')}
10:       if s' ∉ V then
11:          V := V ∪ {s'}
12:          W := W ∪ {s'}
13:
14: return (V, E)
```

If $|\mathsf{reach}(s_I)| = \infty$, Algorithm 1 will not terminate. If $|\mathsf{reach}(s_I)|$ is finite but very large, the algorithm may not terminate successfully. The problem that the reachability graph can be very large or infinite for even simple models is known as the *state explosion problem* [161]. The state explosion problem can be the cause for unsuccessful execution of Algorithm 1 for several reasons. As an example, the available memory can be exhausted causing the algorithm to terminate prematurely or causing the operating system to start swapping internal memory to disk, leading to vastly decreased performance of the algorithm as it it ill-suited for external memory. The problem is basically line 10 of Algorithm 1, as the check whether $s' \in V$ will require access to external memory almost every time. Another problem is that the execution may simply take too long for the result to be interesting, e.g., if the calculation takes two months but the space robot we verify has to launched in one month. If the reachability graph is infinite we must use a method to represent it using only a finite amount of memory, e.g., by representing the graph using graph grammars [140], representing equivalence classes of states of the real reachability graph [26,92], or by using a coverability graph [52,97] instead of a reachability graph. The first two ways of representing infinite reachability graphs can be used for any formalism, whereas the coverability graph can only be constructed for Petri nets.

The idea of the coverability graph is based on the observation that transitions of Place-transition Petri nets are monotone, i.e., that adding more tokens do not inhibit the execution of transitions or alter the effect of executing transitions[1]. Thus, if we reach a state, s', which has at least the same number of tokens on all places as a previously visited state, s, written $s' \geq s$, all transitions enabled in s will also be enabled in s' (this is the definition of monotonicity). Thus it is possible to execute the transition sequence leading from s to s' an arbitrary number of times, each time producing more tokens. We can replace the number of tokens on places in s', which contains strictly more tokens than

[1]Certain extensions of PT-nets do break monotonicity, however, e.g., inhibitor arcs [21] that checks for absence of tokens, or bounds on places that prevent adding more than a fixed number of tokens to places.

in s, with infinity (∞), representing that it is possible to generate an arbitrary number of tokens on these places. As an example, in the case of the PT-net model of a network protocol in Fig. 1.7, we see that by executing Send Data we reach a state where the number of tokens on all places except Network 1 are the same as in the initial state. On Network 1 we have one more token after executing Send Data, so we replace the number of tokens on Network 1 with ∞, signifying that by executing Send Data an arbitrary number of times, we can produce an arbitrary number of tokens on Network 1. The coverability graph can be used for, e.g., determining upper bounds on the number of tokens on each place and thus whether the PT-net is bounded, so it is possible to use the coverability graph to determine if the reachability graph of a PT-net is finite. The coverability graph method cannot immediately be used for CP-nets as it is not possible to define a canonical ordering of states which makes transitions monotone and still guarantees that the coverability graph is finite as the types of places can be infinite.

In the rest of this chapter we will only consider finite but large reachability graphs. We are thus interested in reduction techniques for storing reachability graphs efficiently.

2.2 Reduction Techniques

Reduction techniques for finite reachability graphs basically fall into two categories: algorithms for explicit representation of the reachability graph and algorithms for symbolic representation of the reachability graph. Reduction techniques for explicit reachability graph analysis, basically fall into four categories: methods that explore only a subset of the reachability graph directed by the verification question [42, 134, 160]; methods that use external storage to store the set of visited states [153, 156]; methods that delete states from memory during reachability graph exploration [25, 60]; and methods that store states in a compact manner in memory [T1, T2, 57, 76, 93]. Symbolic reachability graph analysis typically use binary decision diagrams (BDD) [12] or multi-valued decision diagrams (MDD) [96] to store states, or represent each state of the system as a propositional formula and rely on a SAT-solver [148], e.g., MiniSAT [43] or HyperSAT [83], to do bounded model checking [8]. We will first take a look at some symbolic techniques and then turn to examples of reduction techniques from each of the four categories.

Symbolic reachability graph analysis using BDDs relies on representing each state of the system as a bit-vector. A set of bit-vectors can be efficiently represented using a finite automaton accepting exactly the bit-vectors in the set, and BDDs are one way to represent such automatons efficiently. BDDs reduce the memory needed to store each state by sharing the representation of common parts of the bit-vectors. If we cannot represent the state as a bit-vector, e.g., in the case of PT-nets where we have no a priori bound for all places, we can use MDDs, which are able to represent strings of integers using similar techniques as BDDs.

The basic idea of bounded model checking is to encode all executions of the system after executing k transitions as a boolean formula M_k. We conjunct this with a formula, $\neg\varphi_k$, which states that a property φ does not hold after executing k transitions. If (and only if) $M_k \wedge \neg\varphi_k$ is satisfiable, an execution of length k which does not satisfy φ is found. We assume that a state of the system can be encoded as a vector, s, consisting of n boolean variables, $s[0], \ldots, s[n-1]$. We let s_0 be the initial state and let $I(s_0)$ be a propositional formula encoding the initial state. We let $T(s_{i-1}, s_i)$ be a formula that is satisfiable if there is a

transition leading from the state s_{i-1} to the state s_i. The formula M_k is then expressed as $M_k = I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i)$. If we take $k = 2^n$, we are guaranteed to reach all possible states, as at most 2^n different states can be encoded using n boolean variables, and after executing 2^n steps we are thus guaranteed to have reached either states with no successors or previously seen states and therefore we have discovered a loop in the reachability graph. If we want to model check an invariant property, $\mathcal{I}$, we can take $\varphi_k = \bigvee_{i=0}^{k} \neg\mathcal{I}(s_i)$. Bounded model checking relies on solving the problem of whether a propositional formula is satisfiable. This is referred to as the SAT problem and is a well-known NP-hard problem [55]. This means that no known algorithm can solve the problem in time polynomial in the number of variables. There exist, however, very efficient heuristics, and the advantage of bounded model checking is that we check all states reachable in k steps in a single iteration. Bounded model checking is only applicable if we can encode the state of the system as a vector of boolean variables, which is, e.g., the case for bounded Place-transition Petri nets when the bound is known in advance.

The main problem with symbolic model checking is that it works best if all reachable states can be represented using a bit-vector or vector of boolean variables of the same size. This is not the case for CP-nets, which is why we are mainly interested in explicit reachability graph analysis. For example the CPN model of a network protocol in Fig. 1.8 contains integers, which in principle can grow unbounded as well as a list of packets (on In Buffer) which is also in principle able to grow arbitrarily long. Even if a bound can be found, it will often be very large as CPN states are often hundreds of bytes, so symbolic model checking would need to deal with thousands of bits/boolean variables, rendering the methods virtually useless for CP-nets.

Now let us instead consider explicit reachability graph analysis. Depending on the property we want to check, we may not need to explore all reachable states to be able to provide correct answers. One way to only explore a subset of all states is to use a partial order reduction [28, 136]. Partial order reduction exploit the fact that some transitions can be executed in any order yielding the same result. As an example, consider a state, s. If we can execute the transitions a and b *concurrently*, i.e., if $s \xrightarrow{a} s' \xrightarrow{b} s'''$ and $s \xrightarrow{b} s'' \xrightarrow{a} s'''$. If we are only interested in the behaviour after executing both transitions, we only need to consider one of the two execution sequences. This allows us to check, e.g., whether a system has any *dead-locks*, i.e., states with no enabled transitions. As the number of possible execution sequences grow exponentially as a function of the number of concurrently enabled transitions, only exploring one (or a few) of them yields a huge optimisation. The problem is, of course, identifying such sets of transitions. For PT-nets, we can check if all tokens required by a set of transitions are available by adding the number of tokens consumed for all transitions of the set. If all tokens are available the transitions can be executed concurrently. For non-monotone formalisms the analysis is more complex, as we also have to check whether the execution of one of the transitions in the set can inhibit the enabling of some of the others. There are numerous variants of partial order reductions, such as ample sets [134, 135], persistent sets and sleep sets [58, 59, 171], and stubborn sets [160]. Another way to only explore parts of the reachability graph is to use a weight function, which assigns higher weight to transitions that are likely to lead to states violating the property we wish to check. This is known as directed model checking [42], and such weight functions can either be provided manually by the user or, in some cases, computed automatically.

External memory algorithms [153, 156] for reachability graph analysis basically store states on disk sorted according to some ordering of the states. Storing states and checking whether states are already stored are batched, minimising the number of disk accesses required. In addition, an in-memory cache is used to further minimise the number of disk accesses required. While such algorithms are interesting, computers today often have enough memory available, counting in gibi-bytes on laptop computers to hundreds of gibi-bytes on large servers, that by just representing states more efficiently we can analyse systems for which filling internal memory would take weeks or months, in particular when analysing CP-nets, where calculating enabled transitions can be very time-consuming.

Among methods which delete states from memory during exploration are the state caching [60] and sweep-line methods [25, 104]. State caching basically performs a depth-first traversal of the reachability graph. Rather than storing all states of the reachability graph, only the states on the depth-first stack are guaranteed to be stored. If enabled transitions can be processed in a deterministic way, this will terminate whenever the reachability graph is finite. It is possible that the successors of certain states are explored more than once, however. This happens for states with more than one transition leading to them, i.e., if $s' \xrightarrow{a} s$ and $s'' \xrightarrow{b} s$ for $s', s'' \in \mathsf{reach}(s_I)$ and $(s', a) \neq (s'', b)$. In that case s will be explored more than once. In order to minimise the number of re-explorations, some states are cached in memory, even if they are not on the depth-first stack. Several methods [19, 40, 56] aims at finding clever ways to decide which states should be kept in memory and which should be discarded. Another method for intelligently removing states from memory during analysis is the sweep-line method, which uses a specification to detect when a state will not be encountered again. We go into more detail about the sweep-line method in Sect. 2.2.1, as it is needed to understand the summary of the paper [T1] in Sect. 2.3.

Several algorithms for storing states more efficiently exist, some are dependent on the formalism used and some are independent of the formalism used. For CP-nets, we can use an approach similar to BDDs for storing states, namely storing states in a tree sharing common parts of the state [24]. The idea is to observe that CP-nets are split into places and that the tokens on one place can also be encountered on other places or on the same place in other states. Furthermore the effects of transitions on CP-nets are usually local, meaning that only a few places are modified when an enabled transition is executed. By storing identical marking of places (multi-sets of values) only once, we can thus obtain a reduction in the memory required to store the state of a CPN model. The representation of a state just refer to the correct marking of each place. This can be used with the network protocol to share the markings of Send ID and Receive ID as well as the empty markings of the two network places. Furthermore, CP-nets are extended with a simple module concept, and the locality of transitions means that often only markings of places in one or a few modules are changed. By furthermore representing the state of each module separately, it is possible to re-use the representation of all unchanged modules. This method is implemented in CPN Tools [C1, 33]. Bit-state hashing [76] is a formalism-independent approach to storing states efficiently. Bit-state hashing uses a hash function to compute a hash value for each state. This hash value is then used as index in a bit-array (modulo the size of the array) to set a bit indicating a state with that hash value has been encountered. If multiple states have same hash value this will lead to a *hash collision*, i.e., two different states are considered the same because they have the same hash value. To reduce this

problem one can use more than one hash function or a linear combination of two or more independent hash functions using double hashing [38]. Hash compaction [155, 172], like bit-state hashing, applies a hash-function to each state. Instead of using the hash value as index in an array, the hash value itself is stored. Hash-compaction will be discussed in further detail in Sect. 2.2.2, as the ComBack method described in the summary of the paper [T2] in Sect. 2.4 builds on hash compaction.

2.2.1 The Sweep-Line Method

The sweep-line method [25, 104] introduced by Christensen, Kristensen, and Mailund is an example of a method that deletes states during the analysis. The idea is to introduce a *progress measure* assigning to each state a *progress value*, $\psi : S \rightarrow \mathrm{N}$. In fact, the progress measure can assign progress values from any partially ordered set, but for simplicity we will here assume that we use integers as progress values. In the basic sweep-line method from [25] the idea is to require that if $s \rightarrow s'$ then $\psi(s) \leq \psi(s')$. The progress measure is thus a syntactical way to recognise whether a state s' is reachable from s (if $\psi(s') < \psi(s)$ it is not). In the network protocol example from Fig. 1.8, we can let the progress value of each state be the sum of the Send ID and Receive ID counters. In Fig. 2.1 we have written the progress value of each state as a large number to the upper left of each state. Each state is represented by the value of the token on the Send ID place, the sequence numbers of the packets in Network 1, the number of tokens available on Limit, the sequence numbers of the packets on In Buffer, the value of the token on Receive ID, and the sequence numbers of the packets on Network 2. The initial state is marked by a red background. The progress value of each state is thus the sum of the numbers next to Send ID and Receive ID. Transitions are represented by an abbreviated version of their name and the sequence number of the packet being processed. We note that in this case $s \rightarrow s'$ implies $\psi(s) \leq \psi(s')$ for all reachable states. We explore the reachability graph by always picking states with the lowest progress values first. This means that we can safely delete states with lower progress values because of the contraposition of the requirement for a progress measure, namely that if $\psi(s) < \psi(s')$ then $\neg s \rightarrow s'$, which can be extended to that if $\psi(s) < \psi(s')$ then $\neg s \rightarrow^* s'$. Conceptually, the progress measure defines a sweep-line, so that states behind the sweep-line have all been processed and we know that none of the currently unexplored states will have transitions leading to states behind the sweep-line, so they can safely be removed from memory. In Fig. 2.1 the thick arrow below the states shows the direction we explore the reachability graph. If we draw a vertical line, like the one between the states with progress values 3 and 4, we notice that at no point do transitions cross the sweep-line from right to left (except that states with progress measure 4 use 2 columns for easier display). The basic sweep-line algorithm is given in Algorithm 2. The changes from Algorithm 1 is that we in line 6 select one of the states with the smallest progress value rather than an arbitrary state, we remove states from V with lower progress measure than any state in W in line 13, and we remove any edges that are connected to states that have been removed in line 14. The algorithm can be implemented by representing W using a priority queue with ψ as the priority function. Garbage collection can either be done each time we select a state s in line 6 with a higher progress value than in the previous iteration or every, say, 1000^{th} iteration depending on how V is implemented. If we create a double representation of V using a hash table and a priority queue with φ as priority, we can perform garbage collection each time we increase the progress value without an unreasonable

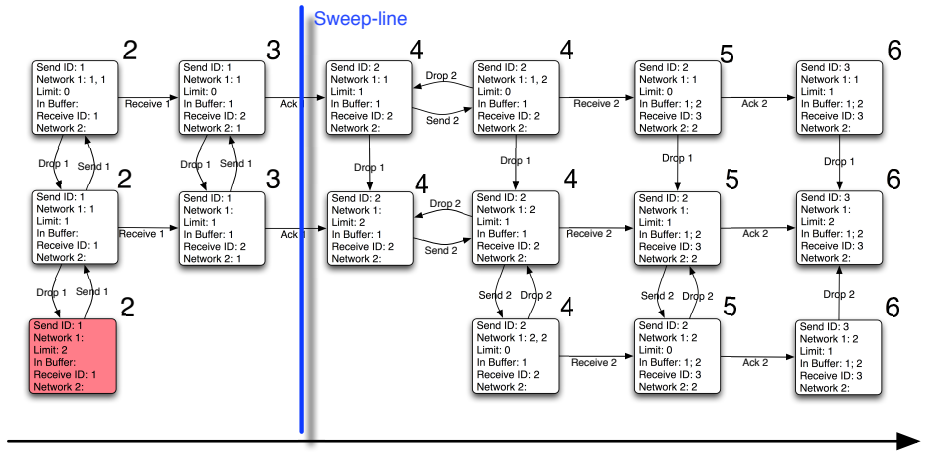


Figure 2.1: The reachability graph of the network protocol with progress values assigned to each state and a sweep-line drawn between states with progress values 3 and 4.

penalty in time. If V is just represented as a has table, how often we do garbage collection needs to be balanced between the cost of traversing all of V against the additional memory required to store additional states that can safely be garbage collected.

Algorithm 2 The basic sweep-line method for reachability graph traversal.

Require:
$\mathcal{LTS} = (S, T, \Delta, s_I)$ a labelled transition system,
$\psi : S \to \mathbf{N}$ a progress measure

1: $V := \{s_I\}$
2: $W := \{s_I\}$
3: $E := \emptyset$
4:
5: **while** $W \neq \emptyset$ **do**
6: Select an $s \in W$ s.t. $\forall s' \in W.\psi(s') \geq \psi(s)$
7: $W := W \setminus \{s\}$
8: **for all** t, s' such that $s \xrightarrow{t} s'$ **do**
9: $E := E \cup \{(s, t, s')\}$
10: **if** $s' \notin V$ **then**
11: $V := V \cup \{s'\}$
12: $W := W \cup \{s'\}$
13: $V := \{s \in V \mid \exists s' \in W.\psi(s') \leq \psi(s)\}$
14: $E := \{(s, t, s') \in E \mid s, s' \in V\}$
15:
16: **return** (V, E)

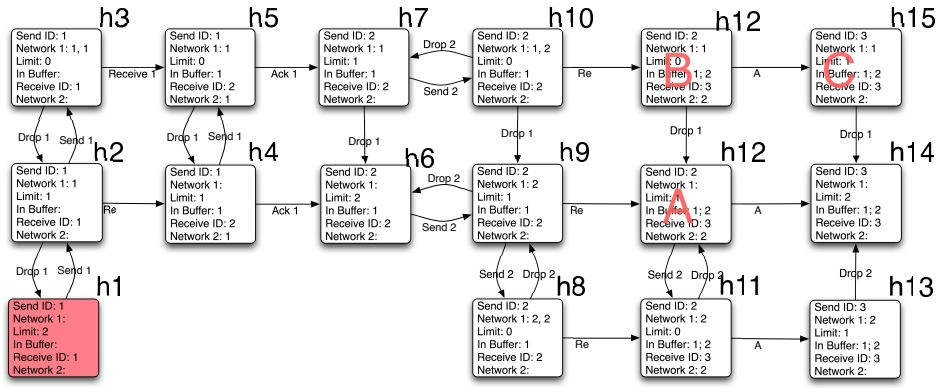
Unfortunately the property that $s \to s' \implies \psi(s) \leq \psi(s')$, i.e., that the progress measure is *monotone* does not hold for many interesting systems, such as reactive systems, unless we choose a trivial progress measure assigning the same progress value to all reachable states. The trivial progress measure does not yield any reduction in the number of states stored. To overcome this, the sweep-line method has been extended by Kristensen and Mailund in [104] to also handle systems where we may have $s \to s' \wedge \psi(s) > \psi(s')$. Edges satisfying

this property are called *regress edges*. By traversing the reachability graph multiple times, each traversal called a *sweep*, the sweep-line method is able to cope with regress edges. The idea is to start from the initial state in the first sweep and in all the following sweeps use the destinations of regress edges found in the previous sweep as starting points. Furthermore, we never remove destinations of regress edges from memory. The major advantage of the sweep-line method is that if the progress measure is good, i.e., if it separates the reachable states into many equivalence classes and yields few regress edges, only a fraction of the reachable states are kept in memory at any time. How to find good progress measures are the topic of much research. As an example Schmidt [151] use transition invariants of PT-nets to automatically synthesise efficient progress measures, and Vanit-Anunchai, Billington, and Gallash try to assist the user in manually obtaining good progress measures by counting the number of states in each class of states with the same progress value [165].

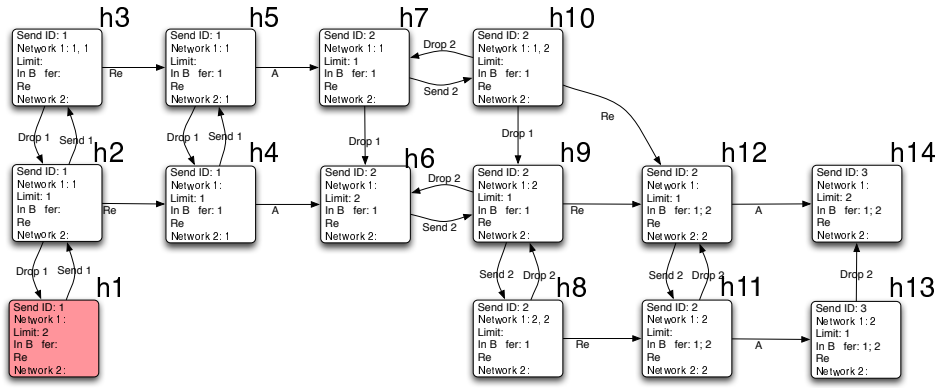
2.2.2 Hash Compaction

Hash compaction [155, 172] introduced by Wolper and Leroy uses a hash function, $H : S \to \{0,1\}^w$, to compress states to w bits before they are stored. As an example, in the network protocol in Fig. 1.8, to represent the state of the system, we would need 12 integers to store each state of the system (one for Send ID, Receive ID, and Limit, one to indicate how many packets and two to specify which packets are on either of Network 1, Network 2, and In Buffer), using 48 bytes assuming that 32 bits are used to represent each integer. By using a hash function generating 32 bit hash values, we would only use 32 bits or 4 bytes to store each state. The algorithm for reachability graph analysis using hash compaction is the same as the algorithm for basic reachability graph analysis, namely Algorithm 1. The only difference is that the checks in line 10 and the adding of nodes in line 11 are implemented in a different way—this actually holds for any algorithm which implements a more efficient state representation. For hash compaction we would check whether $H(s) \notin V$ (in line 10) and replace line 11 by $V := V \cup \{H(s')\}$.

The major caveat of hash compaction is that hash collisions may lead to not exploring all reachable states, as we may incorrectly conclude that a state s' has already been visited if we have visited a state s, whose compressed state descriptor $H(s)$ is equal to the compressed state descriptor of s', $H(s')$. Say we have a hash function assigning hash values $h1$–$h15$ to the states of the network protocol. In Fig. 2.2(a) we have written the hash values assigned to each state to the upper left of the states. If we assume that the state marked with a big A inside is discovered before the state marked B, we will believe we have already seen state B, and not process it further, so the state C will never be discovered. In fact, the reachability graph as explored using hash compaction will look like the one in Fig. 2.2(b). If we consider it an error to have received and acknowledged all packets successfully, yet still have an outstanding copy of the first packet, analysis using hash compaction would (in this case) not discover the error as state C is not explored. The hash compaction method can be improved by using more than one hash function, but the basic problem persists, namely that the method is incomplete in general. In [T2] we introduce the ComBack method, which improves hash compaction by adding a means to discover hash collisions on-the-fly during the traversal, making the method complete as well as sound.



(a)



(b)

Figure 2.2: Reachability graphs for the network protocol as seen when using hash compaction.

2.3 Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method [T1]

The major disadvantage of the sweep-line method is that at no time during exploration do we have a complete representation of the entire reachability graph in memory (unless we use the trivial progress measure assigning the same progress value to all reachable states, in which case the method yields no optimisation), so it is only possible to decide invariant properties. If we want to check more complex properties, such as liveness properties using Linear Temporal Logic (LTL), we will need a representation of the reachability graph in memory or the ability to perform depth-first traversal of the reachability graph as LTL can be checked by calculating strongly connected components of the reachability graph using, e.g., Tarjan's algorithm [157], or on-the-fly using nested depth-first traversal [74] of the reachability graph as described by Holzmann. Neither of these methods are immediately possible in combination with the sweep-line method. Tarjan's algorithm cannot be used as it requires that we have a representation of the reachability graph in memory (or that we are able to generate the graph in a depth-first manner), and nested depth-first

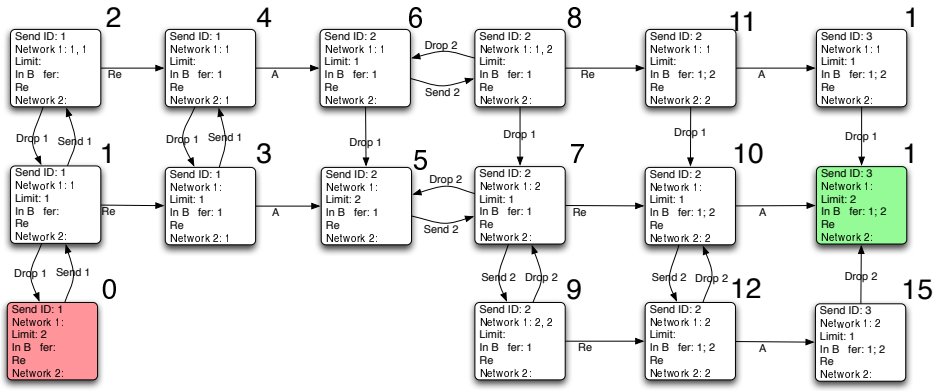
traversal of the graph is not usable as the sweep-line imposes a certain order of traversal depending on the reachability graph in order to perform well. If we use the basic sweep-line method it is possible to check LTL as all states in a strongly connected component will need to have the same progress value. The basic sweep-line method yields no optimisation for reactive systems, however, and it may often be possible to devise a better progress measure if we allow a few regress edges. Our paper [T1] uses the sweep-line method to construct a near memory-optimal representation of the reachability graph, so we can use either Tarjan's algorithm or nested depth-first traversal to check, e.g., liveness properties.

The most efficient representation of $|S|$ states use $\lceil \log_2 |S| \rceil$ bits to store each state[2]. Often the encoding actually used is not even this efficient, so even more than the required $\lceil \log_2 |S| \rceil$ bits are used to store each state. In the network protocol example, we would use 48 bytes (to represent 12 integers) or 384 bits to store each state. Only $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits are actually needed to distinguish between the $|\mathsf{reach}(s_I)|$ reachable states, however. The idea of [T1] is that the number of reachable states is often much smaller than the number of syntactically possible states, $|\mathsf{reach}(s_I)| << |S|$, so we map representations of states from S (the full state descriptors) into bit-vectors of size $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ (the condensed representation) in a way so that we can later analyse the reachability graph. In the network protocol we only need to use $\lceil \log_2 16 \rceil = 4$ bits for each state, using only around 1% of the memory used for our naive representation of each state. This representation is realised by representing each reachable state as a number $0, \ldots, |\mathsf{reach}(s_I)| - 1$, and using a standard successor-list representation of the reachability graph. Such numbers have no relation to the full state descriptor, so we need to keep the full state descriptors as long as needed to recognise previously seen states. In Fig. 2.3(a) we see the reachability graph of the network protocol in Fig. 1.8. We have assigned to all states a state number, written to the upper right of the state. A successor-list representation of the reachability graph can be seen in Fig. 2.3(b). For each node we store a pointer to a list of all successors. The list is preceded by the number of successors, and contains a list of pairs with the transition and the number of the state it leads to. As an example, we can see that the state with number 1 has 3 successors. One successor is reached by executing Drop 1 and leads to state number 0, and the other successors are reached by executing Send 1 leading to state number 2 respectively executing Receive 1 to state number 3.

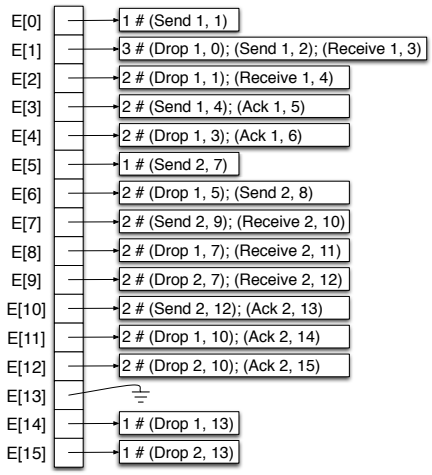
If we assume that the transition relation is *deterministic*, i.e., if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$, this structure can be traversed using Algorithm 3. The idea is to traverse the graph according to the condensed representation (the numbers), and calculate the full state descriptors during traversal using the transition information. We use the fact that the transition relation is deterministic to calculate the successors in line 10 of the algorithm. It is easy to change Algorithm 3 to check Computation Tree Logic (CTL) as in [27, Sect. 4.1] by adding a table of sub-expressions of the formula to check, indexed by $0, \ldots, \lceil \log_2 |\mathsf{reach}(s_I)| \rceil - 1$ so it is possible to calculate a fix-point of satisfied formulae in each state. We can also extend the algorithm to use nested depth-first search [74], so it can be adapted to check Linear Temporal Logic (LTL).

One problem is, of course, to recognise when a full state descriptor is no longer needed, i.e., when we will never encounter it again. We use the sweep-line to delete full state descriptors from memory when they are no longer needed. Another problem is that when we start the generation we do not know

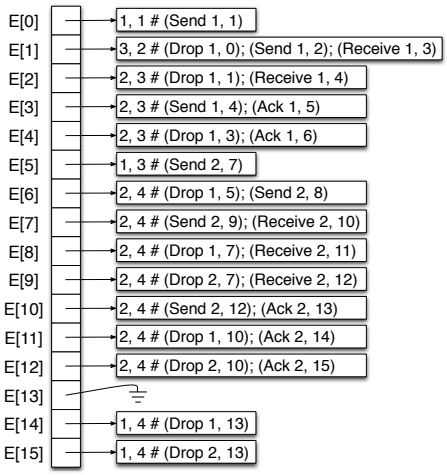
[2]Assuming that $|S| < \infty$; if $|S| = \infty$ we would use a variable-length encoding.



(a)



(b)



(c)

Figure 2.3: The reachability graph of the network protocol with progress values assigned (a) and two successor-list representations of the graph (b) and (c).

$|\mathsf{reach}(s_I)|$, so we do not know how many bits to use for each state. To circumvent this problem, we simply use as many bits as required to store all successors (which is known at the time we store successors) and store this length as well.

The algorithm works by defining a function $\mathsf{idx}_M : S \to \{0, \ldots, |\mathsf{reach}(s_I)| - 1\}$, mapping full state descriptors to state numbers. This function can, e.g., be implemented as a hash table. Whenever we encounter a new state we assign it a new state number and add it to idx_M. Consider, e.g., Fig. 2.3(a). If we are exploring state 1 and the state 0 is already in idx_M, we will encounter two new states, which we assign numbers 2 and 3. As we assign numbers to states when they are first discovered, we will always know the maximum state number of all successors when processing a state. In the case of state 1, this number is 3, so by using $\lceil \log_2 3 \rceil = 2$ bits, we can store all successors of state 1. The structure in Fig. 2.3(c) shows how we can represent the reachability graph by extending the header to also include how many bits are used to store each successor state. As an example we see that state number 1 has 3 successors, each represented using 2 bits. One of the states is reached by the transition Drop 0 and has number 0. When we have processed state 2 and move on to state 3, we notice

Algorithm 3 Depth-first traversal of the condensed reachability graph

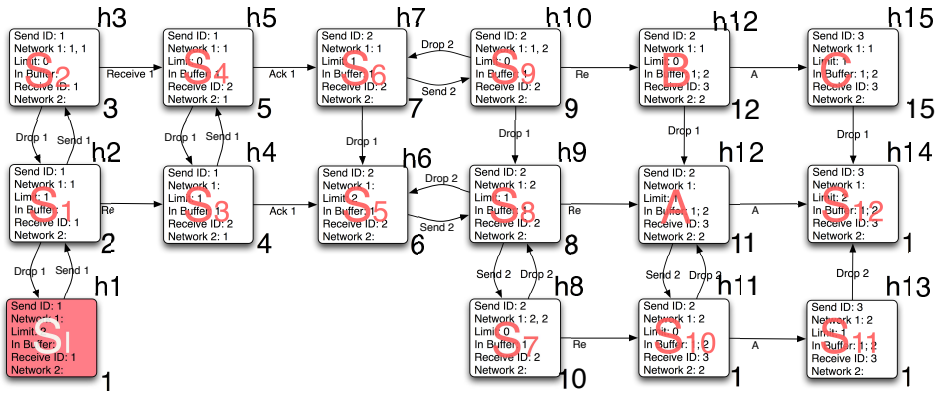
Require: E a successor-list representation of a reachability graph

```
 1: V := ∅
 2: DepthFirstTraversal(0, s_I)
 3:
 4: proc DepthFirstTraversal(i, s) is
 5:   if i ∈ V then
 6:     return
 7:   {analyse s here}
 8:   V := V ∪ {i}
 9:   for all (t, i') in E[i] do
10:     Let s' be such that s -t-> s'
11:     DepthFirstTraversal(i', s')
```

that there is no need to store the idx_M mapping for states 0 to 2 as we will never encounter them again. We know that because we can look at the reachability graph in Fig. 2.3(a), but using the sweep-line method, the algorithm is also able to realise that, as state 3 has progress value 4 (using the same progress measure as in Fig. 2.1, namely the sum of Send ID and Receive ID) and the states 0 to 2 have progress value 3. We therefore remove the mapping of the full state descriptors of states 0–2 from idx_M and proceed calculating successors of state 3 and assigning them state numbers. If the progress measure is not monotone, it is possible that we delete a full state descriptor from idx_M before we are done using it. This will lead to the state being assigned a new number, so the reachability graph constructed using this algorithm may actually be an *unfolding* of the original reachability graph. The unfolding is shown to be bisimilar [124] to the original reachability graph by Mailund in [118, Chap. 13], so CTL* and in particular LTL and CTL is preserved as shown by Clarke, Grumberg, and Peled in [27, Chap. 12]. The full algorithm can be seen in Fig. 5.3 on page 86.

2.4 The ComBack Method—Extending Hash Compaction with Backtracking [T2]

As can be seen in Fig. 2.2, the hash compaction reduction method may lead to not exploring all reachable states if there exist two states, $s \neq s'$, with the same compressed state descriptor, $H(s) = H(s')$, as is the case with states A and B in Fig. 2.2(a). The ComBack method circumvents this by storing enough information that we are able to realise that the states s and s' are actually different even though $H(s) = H(s')$. Like the method in the previously discussed paper [T1], we will represent each state as integers, in this case $1, \ldots, |\mathsf{reach}(s_I)|$. Furthermore, we use a hash function H to generate compressed state descriptors for each state. In Fig. 2.4(a) we have shown the reachability graph of the network protocol from Fig. 1.8 and assigned each state a number $1, \ldots, |\mathsf{reach}(s_I)|$ and a compressed state descriptor $h1, \ldots, h15$. Additionally, we have assigned each state a name, s1—s11, A, B, and C, to have a brief way of referring to the full state descriptor of each state. As an example, we see that the states A and B have the same compressed state descriptor, $h12$. For the sake of the description of the method, we will assume that the transition relation given is deterministic, i.e., that if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$.



(a)

h1	1
h2	2
h3	3
h4	4
h5	5
h6	6
h7	7
h8	10
h9	8
h10	9
h11	13
h12	11, 12
h13	16
h14	14
h15	15

(b) State table

1	
2	1, Send 1
3	2, Send 1
4	2, Receive 1
5	3, Receive 1
6	4, Ack 1
7	5, Ack 1
8	6, Send 2
9	7, Send 2
10	8, Send 2
11	8, Receive 2
12	9, Receive 2
13	10, Receive 2
14	11, Ack 2
15	12, Ack 2
16	13, Ack 2

(c) Backedge table

Figure 2.4: The reachability graph of the network protocol with compressed state descriptors and state numbers (a), the state table (b) and the backedge table (c) used to represent the reachability graph using the ComBack method.

The method can be extended to also deal with non-deterministic transition relations; for more details refer to Sect. 6.5 on page 103.

The ComBack method works by storing a mapping from compressed state descriptors to state numbers, called a *state table*, which maps a compressed state descriptor into all encountered state numbers with the corresponding compressed state descriptor. The state mapping for the reachability graph of the network protocol can be seen in Fig. 2.4(b). Furthermore we maintain a spanning tree from the initial state to all explored states by storing, for each state number n' corresponding to a state s', the number, n of a predecessor state s and a transition t such that $s \xrightarrow{t} s'$. This information is stored in a data-structure called the *backedge table*. A possible backedge table for the reachability graph of the simple network protocol can be seen in Fig. 2.4(c). Here we see, e.g., that state number 2 can be reached from state number 1 via the transition Send 1.

Using the backedge table it is possible to reconstruct the full state descriptor for each state number. Say we want to reconstruct the full state descriptor for the state with number 11. We look up state number 11 in the backedge

table and obtain 8, Receive 2, meaning we must take a Receive 2 transition from the state with number 8 to reach the state with number 11. We now look up state number 8, and get 6, Send 2. We continue and obtain (4, Ack 1), (2, Receive 1), (1, Send 1). The state with number 1 is the initial state. We can see this as it has no backedges. Now we must execute the transition sequence we have obtained from the initial state in the reverse order, and get $s_I \xrightarrow{Send\ 1} s_1 \xrightarrow{Receive\ 1} s_3 \xrightarrow{Ack\ 1} s_5 \xrightarrow{Send\ 2} s_8 \xrightarrow{Receive\ 2} A$, which is indeed the state with state number 11. We can thus backtrack from any given state number to the initial state and execute all the transitions stored in the backedge table to obtain the full state descriptor corresponding to the state number.

The state table and the backedge table are created as we explore the reachability graph. Whenever we encounter a state, s', using the transition $s \xrightarrow{t} s'$, we calculate its compressed state descriptor, $H(s')$, and look up all state numbers corresponding to that compressed state descriptor in the state table. If no such state numbers exist, we just assign the state a new number and add it to the state table and the backedge table. If there are any such state numbers in the state table, we use the aforementioned technique to re-generate the full state descriptors for each of the numbers. These can then be compared with the full state descriptor of s'. If any of the full state descriptors are equal to s', we have already encountered s', and do not need to proceed. Otherwise we just assign s' a new state number and add it to the state table and the backedge table. All we need to know to add a state to the state table and the backedge table is the number of a predecessor and a transition from the predecessor as well as the highest used state number. The entire algorithm can be seen in Algorithm 5 on page 100.

In addition to the basic algorithm, a number of variants are given in [T2]. One variant is able to also handle non-deterministic transition relations. This variation also makes it possible to only store the number of a predecessor state in the backedge table and omitting the transition information, yielding a memory optimisation at a cost in time. Time-saving variants include shortening of how long we need to backtrack for each state. The lengths of backtracks depend on the traversal policy—in the example we have used a breadth-first traversal yielding optimal backtracks, but if we had used, e.g., depth-first traversal, the backtracks might not be optimal, so shortening backtracks would reduce the time required to reconstruct full state descriptors. Another time-saving variant simply caches some full state descriptors, thus spending a little more memory for increased performance.

2.5 Contribution and Future Work

In this chapter we have taken a look at behavioural verification of formal models including symbolic methods and four categories of explicit reachability graph analysis. We have in particular looked at the existing sweep-line and hash compaction reduction techniques and how our papers have improved upon these methods. In this section, we sum up the contributions made in this field and provide directions for future research.

Our first presented paper, [T1], improves upon the sweep-line reduction technique by using the sweep-line method to construct a near-optimal representation of the reachability graph. This representation makes it possible to use the sweep-line method to check properties that are more complex than invariant properties, e.g., liveness properties in LTL. The algorithm is evaluated on a number of examples in [T1]. Unsurprisingly, the algorithm works

best when the sweep-line method does, i.e., on reachability graphs with a clear notion of progress. One such example is an extended version of the network protocol in Fig. 1.8, where 5–10% of the memory required to construct the full reachability graph is used, and 25–130% of the time is spent. If we consider a model of the dining philosophers problem, it is possible to define a progress measure which separates the reachable states into a lot of classes and only yields few regress edges. Using the number of eating philosophers as progress value, we will store almost all states during the construction, so the memory used for the compact representation is overhead. Compared to the amount of memory used for the full state descriptors, this is negligible, however, and the only real disadvantage is the extra time is spent during construction.

Our second presented paper, [T2], makes the hash compaction reduction technique complete by storing, in addition to a compressed state descriptor, a spanning tree rooted in the initial state. This extra information makes it is possible to resolve hash collisions on-the-fly by reconstructing the full state descriptor when we encounter states with a compressed state descriptor already stored. The paper compares the ComBack algorithm to other algorithms that minimise the amount of memory used to store states, and find that the ComBack method naturally uses more memory than hash compaction, but on the other hand guarantees that all reachable states are explored. The method uses less memory than storing the full state descriptors, and uses around twice as long time. On the other hand, thanks to the lower memory consumption, the method is able to explore reachability graphs that are impossible to explore using the basic algorithm.

In [49] Evangelista and Pradat-Peyre introduce an approach similar to the ComBack method. The method also stores states as pairs of a predecessor and a transition but, compared to the ComBack method, the compressed state descriptor is not stored, and state numbers are merely inserted into a hash table, which can lead to many more reconstructions. Furthermore, [49] has stratified caching [56] built into the algorithm, which makes it less flexible unless we first factor out the caching mechanism, as is done by the ComBack method. In [49] stratified caching with a maximum parameter of 50 is used. This means that a backtrack has length at most 50 and corresponds to caching 2% of the full state descriptors, makes the algorithm use $200\% - 400\%$ of the time used for a basic exploration of the reachability graph. The ComBack method use a simpler caching strategy, namely inserting the mapping from compressed state descriptors to full descriptors into a hash table that does not handle collisions. Using this caching strategy we are able to obtain comparable time results using a cache of only 0.1%–1% of all the states, or approximately 20 times smaller than the cache used in [49]. Furthermore, [49] presents the algorithm solely as a depth-first traversal, whereas the ComBack method is presented in a traversal-independent manner, making the ComBack method easier to combine with other methods. Also, our experimental results show that breadth-first traversal of the reachability graph may be much faster for highly reactive systems where most of the reachability graph end up on the recursion stack.

2.5.1 Future Work

While prototype implementations of the two methods described in this chapter have shown promising performance, neither method has been used extensively in practise. The reason is that one of the methods, the ComBack method [T2], has only recently been published at the time of writing. The other method, the extended version of the sweep-line method [T1], is mainly useful for checking more complex properties, such as liveness using Linear Temporal Logic, and

this does not have easy accessible tool support in tools supporting the algorithm, making real-life applications difficult. In this section we will provide some directions for interesting future work, including some ideas on how to alleviate these problems.

Use the ComBack method in conjunction with other reduction techniques

As mentioned, a strong point of the ComBack method is that it is independent of the traversal type, making it easily adaptable to tasks such as on-the-fly verification of LTL using nested depth-first traversal or CTL model-checking using backwards fix-point calculation. This makes the algorithm well-suited for analysis, and it also makes it easy to combine the algorithm with other reduction techniques, which may impose a certain traversal order. Here it is in particular interesting to combine the method with partial order reduction techniques, which reduce the in-degree of nodes (as high in-degrees often occur when executing concurrent transitions), thereby reducing the number of reconstructions required.

It is also interesting to combine the ComBack method with the sweep-line method. As stated earlier, the sweep-line method and the method for obtaining an efficient reachability graph representation described in Sect. 2.3 work well for reachability graph with a clear notion of progress. We can call such reachability graphs "long", because they often consists of a few long execution traces only. The ComBack method described in Sect. 2.4 works well for "wide" reachability graphs where the graph consists of many short execution traces with little interaction. The sweep-line based method only conserves memory if the progress measure makes it possible to often remove full state descriptors, but the method uses extra time whenever a regress edge leads to an already discovered state because of rediscovery. The ComBack method uses long time reconstructing already visited states, but benefits greatly from a cache mapping compressed state descriptors to full state descriptors. We have only experimented with very simple caching strategies, and our own research as well as that of Evangelista and Pradat-Peyre [49] indicate that the method is very sensitive to caching strategy in terms of how much time is spent. One way to obtain a caching strategy that intuitively should perform well is to use the sweep-line method to define the caching strategy, and cache full state descriptors in front of the sweep-line. Another way to view this combination is that we utilise the ComBack method to test whether destinations of regress edges lead to new or to already discovered states during a run of the sweep-line method. We observe that the successor-list representation of the reachability graph created in our paper [T1] looks very similar to the backedge table of the ComBack method (compare Fig. 2.3(c) with Fig. 2.4(c)). The successor-list representation stores successors and the backedge table stores a predecessor for each state. Both of the tables rely on state numbers. If we extend the successor-list of the sweep-line based method with a predecessor like in the backedge table and introduce a state table like in the ComBack method, we are able to cope with regress edges by, rather than just concluding that they are regress edges and processing them in the next sweep, checking whether we have already encountered the state (by checking the state table and reconstructing states as necessary), and, if the state is new, either schedule it for later processing or process it immediately. As we use the sweep-line method to represent all full state descriptors in front of the sweep-line, we only need to reconstruct destinations of regress edges during the first sweep. As we are able to check whether the destination of a regress edge leads back to a previously unvisited state or

a completely new state, we never need to reconstruct parts of the reachability graph, which was the major caveat of the sweep-line based algorithm. The combined method should therefore be able to analyse systems which only exhibit limited progress, as regress edges no longer lead to a blowup in spent time.

Devise more usable specification language for properties of coloured Petri nets

As mentioned, the new method described in Sect. 2.3 has not been tested extensively due to the lack of reasonable tool support. The current tool, CPN Tools, provides provisional support for checking CTL and support for checking LTL on-the-fly has been experimentally implemented for coloured Petri nets in DESIGN/CPN [37] by Mikkelsen [122] and by the author of this thesis in a model-checker implemented in the BRITNeY Suite. All of these implementations use a textual syntax for describing the temporal formulae and use Standard ML predicates applied to a representation of the state of the model as atomic propositions. This has the disadvantage of requiring that the user is familiar with the complexities of temporal logics and the difficulty of writing often complex predicate functions.

Rather than inventing a new language for specifying properties, one can also just use the modelling formalism itself to specify properties. This has been done in SPIN [77], where properties are stated as so-called never-claims, which is a standard process in the native PROMELA language of SPIN. If the never-claim reaches a final state, it is considered an error. Something similar has been proposed by Petri [139] for Petri nets. Here we add special transitions, called facts, which must never be enabled. We could do something similar for CP-nets by introducing a module containing a place which must never be marked and/or a transition which must never be executed. CP-nets currently have tool support for synchronisation of modules by means of shared places, but support can dually be added for synchronisation by means of shared transitions. Using this it would be possible to create a module representing a scenario which must never happen.

It is also possible to try to define atomic propositions in a simpler way. This idea is partially inspired by Cardelli and Gordon's ambient logic [16] where atomic propositions are stated in a language closely resembling the language of the ambient calculus [17]. For coloured Petri nets something similar could, e.g., be achieved by showing the user a copy of the net/a module of the net. Tokens can then be assigned to places of interest to signify that these tokens must be present on the place in a state for the atomic proposition to hold. For example, in the case of the network protocol in Fig. 1.8, we may want to check if the token [(2, " model"), (1, "Formal")] can ever be present on In Buffer, signifying that the packets have arrived out of order.

Finally, we may want to specify temporal properties in simpler ways than by using a logic. We can, e.g., specify temporal properties using message sequence charts [67] as message sequence charts basically define a partial ordering of events. By annotating message sequence charts with atomic propositions, which must hold between events, users would be able to easily specify even complex temporal properties.

Create test-suite and tools for improvement of reachability graph analysis methods

As can be seen in Table 6.1 on page 107, re-printed from [T2], a lot of experiments have been run in order to validate the usefulness of the ComBack algo-

rithm. The results shown only comprise a small fraction of the total number of experiments, and it is not desirable to have to run these experiments manually.

In [133], Pelánek state that reachability graphs basically come in three variants: random graphs, reachability graphs generated by small academic examples, and reachability graphs generated by realistic/real-life models. Graph-theoretic properties varies for each kind of graph. Often the performance of reduction techniques varies hugely dependent on the structural properties of the reachability graph. For example, the ComBack method works best if the in-degree is small so only few states are reconstructed. This demonstrates that it is important to test methods on several different kinds of models, preferably both academic examples, like the dining philosophers, as well as real-life models.

Furthermore, we would also like to be able to compare results of tests with known good results and compare the execution time and memory consumption as time progresses and the implementation is improved. Both to compare different implementations of the same reduction technique and to compare different reduction techniques.

This implies that we would like a test-suite and supporting tools, which provide a means to automatically run several reachability graph analysis tasks and which preferably provide a means to easily specify such tasks. The test-suite should consist of several formal models, both simple academic examples and real-life models. It should be possible to store results of executions, compare each result to known-good values, and enable exploration of the execution time/memory consumption for different reduction techniques and different implementations.

One such test-suite is being developed within the ASCoVeCo (Advanced State Space Methods and Computer tools for Verification of Communication Protocols) project [3] at the University of Aarhus. The author of this thesis participates in this project and has contributed to the development of the test-suite and tools for running tests.

Improve memory handling

As can be seen in Table 6.1 on page 107, while the ComBack method conserves memory, it uses significantly more than the expected limit of 5 words (20 bytes when a machine word is 32 bits) per state (obtained from Theorem 6.1 on page 103 by using a machine word for each of the mentioned numbers). The theorem does not account for memory used for the state table and backedge table, which will in fact use 2 extra machine words for each state (if implemented as an dynamically extensible array). The state table will also use 2 extra machine words for each state. This yields a total of 9 machine words or 36 bytes for each state, assuming a 32 bit architecture. Yet the best result obtained in Table 6.1 is that 82 bytes is used per state, or more than twice the expected amount of memory required. This is primarily due to the fact that the algorithm is implemented in Standard ML. While Standard ML is a very nice language for specifying algorithms, it is not well-tailored to fine-grained control of memory use. In particular it usually stores a pointer in addition to the data we are interested in, doubling memory used when we primarily store machine words. By implementing the data-structures in C++ and keeping the algorithm implementation in Standard ML, it would be possible to maintain a nice declarative way to describe algorithms, while using a low-level language to use fine-grained control of memory consumption.

Implementing data-structures in a low-level language would also make it possible to implement the condensed representation of [T1] more efficiently.

The current implementation uses a machine word to store each state number, even though the algorithm facilitates using only the number of bits required. A similar trick could be done with the ComBack method from [T2] by, e.g., observing that we know that the number of bits required to store all predecessors of state number 2^n is only n as the predecessor will have a lower number (unless we make path optimisations). This allows us to drastically reduce the amount of memory needed to store the backedge table. We can also use Geldenhuys and Valmari's very tight hashing [57] to represent the state table of the ComBack algorithm more efficiently. All of these optimisations are not feasible when the data structures are implemented in Standard ML, as it is very expensive to pack and unpack data in order to store data that is not word aligned in Standard ML.

Visualisation of error traces for property violations

The BRITNeY Suite [T3, C2], which is described in the next chapter, supports visualisation of traces to violations of invariant properties and liveness properties formulated using LTL by means of simple message sequence charts. Such violations are quite easy to visualise, as a violation of an invariant property can be proved by providing a trace from the initial state to a state not satisfying the invariant. Violations of LTL properties are rather simple to visualise as well, as they can be proved by providing a trace to a loop not satisfying the property. The latter can be visualised by two message sequence charts, one showing the trace to the loop, and one showing the loop.

Violations of properties formulated using CTL are more complex, however. The reason is that a proof of a violation is an annotated version of the reachability graph. Such a proof can of course visualised as a huge graphical graph, but as soon as the graph contains more than a few dozen nodes, this becomes impractical. Instead, we propose another way to convince users that properties hold/do not hold. The idea is that if the users need convincing that the formula does not hold, it is because he thinks it does hold. In Sect. 3.5 we provide more details of how this could be done by letting a user try to prove sub-formulae of the system by choosing some transitions and letting the computer choose other transitions in a way such that it is impossible for the user to ever arrive at a proof of the property.

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}
(& \{\ \ ID, PACKET, PACKETS, PACKETxPACKETS\ \ \}, \\
& \{\ \ P_1, P_2, P_3, P_4, P_5, P_6\ \ \}, \\
& \{\ \ T_1, T_2, T_3, T_4\ \ \}, \\
& \{\ \ 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\ \ \}, \\
& \{\ \ (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\ \ \}, \\
& \{\ \ (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\ \ \}, \\
& \{\ \ 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`[]\ \ \}\ \ )
\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

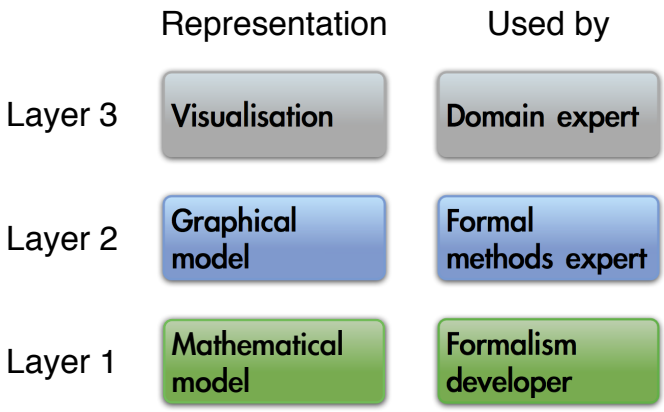


Figure 3.2: Three layers of use of formal models.

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.

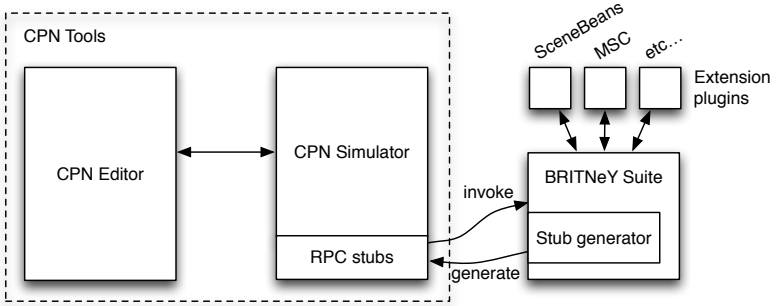
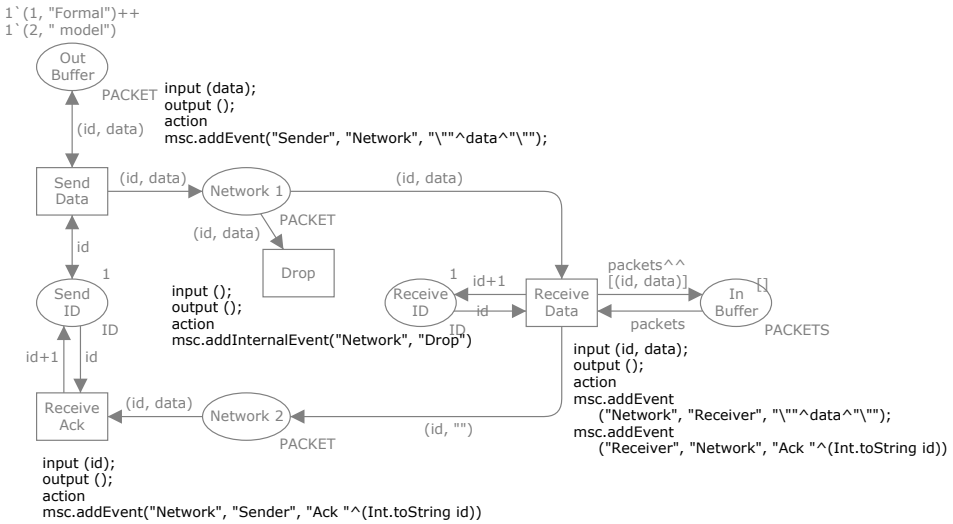


Figure 3.3: Architecture of the BRITNeY Suite when used with CPN Tools

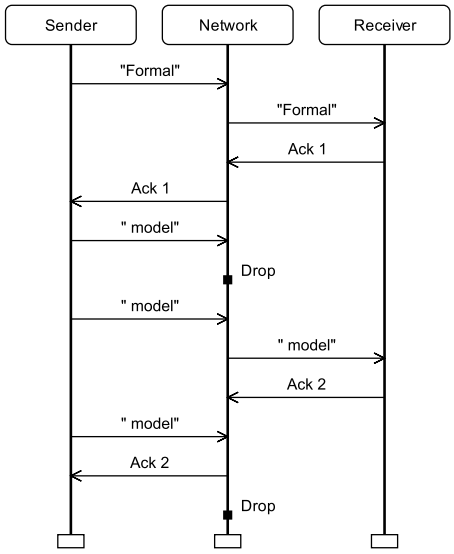
Visualisations are tied to models like in MIMIC/CPN, by annotating the model and calling functions driving the visualisation. Whenever a transition is executed, the corresponding code is executed, which may invoke a stub and consequently a function in the visualisation tool. Consider, e.g., the model 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-

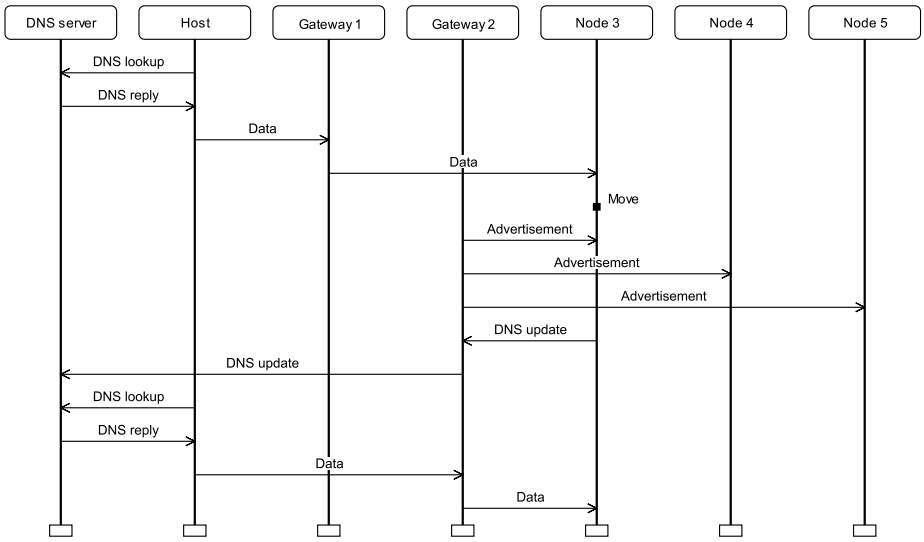


Figure 3.5: A visualisation of an interoperability protocol for mobile ad-hoc networks using message sequence charts.

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

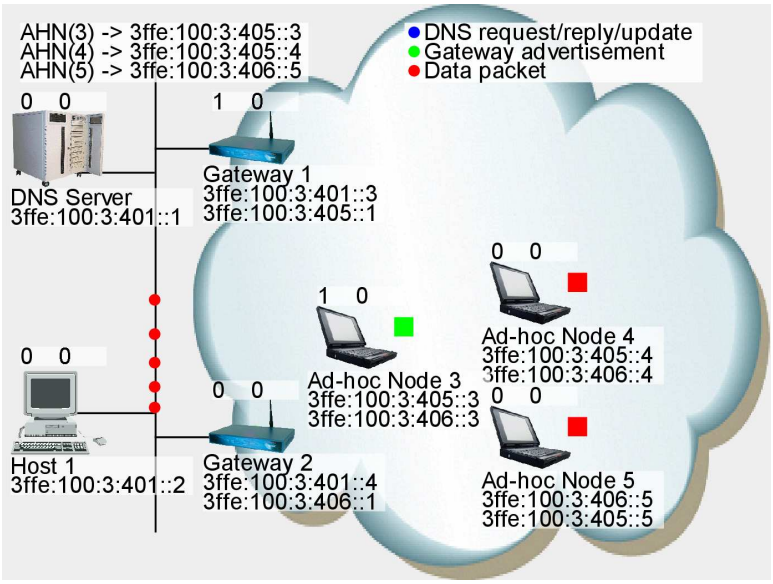


Figure 3.6: A cartoon-like visualisation of an interoperability protocol for mobile ad-hoc networks based on network diagrams.

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

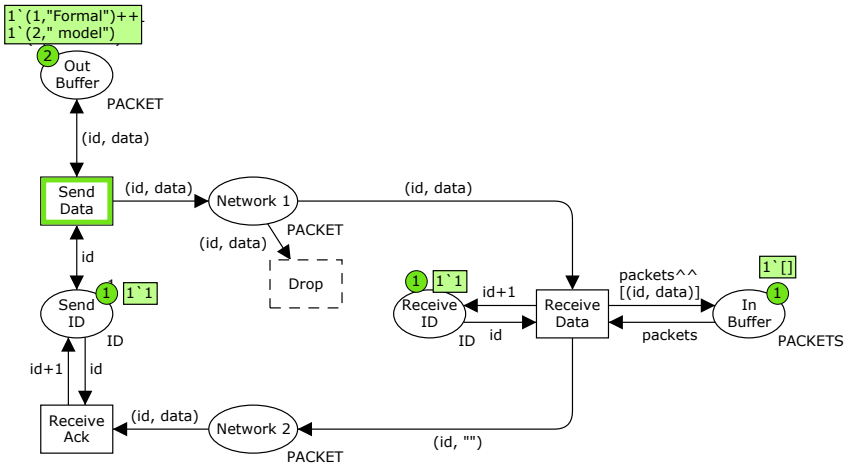


Figure 3.7: A formal model of a simple protocol modelled as a game coloured Petri net.

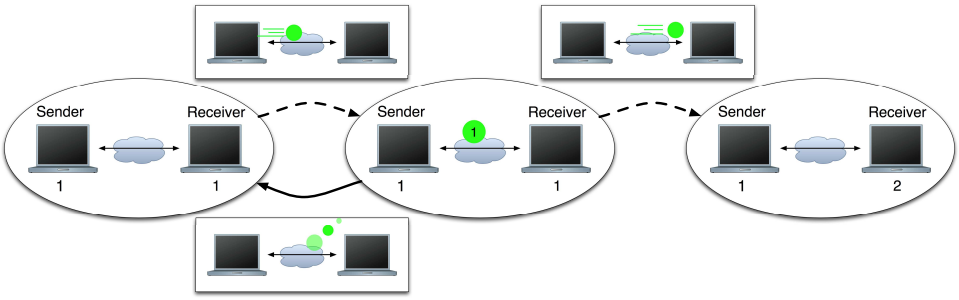


Figure 3.8: Fragment of a visualisation of a simple network protocol as a labelled transition system.

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.

Chapter 4

Summary

This chapter sums up the work done as part of this thesis as well as applications of the work conducted and directions for future work. The contributions of the work is summarised in Sect. 4.1, applications by the thesis author and others of the tools and methods developed as part of the thesis are summarised in Sect. 4.2, and interesting directions for future research are summarised in Sect. 4.3. For more detailed outlines of applications and future work, please refer to Sects. 2.5 and 3.5.

4.1 Contributions

This section summarises the contributions made as part of this thesis. As indicated by the structure of the thesis, work has been done within two areas: behavioural verification of formal models by means of reachability graphs and behavioural visualisation of formal models. The main contribution within the field of reachability graph analysis of formal models consists of improving algorithms for efficient storage of reachability graphs [T2, T1]. The main contributions within the field of visualisation of formal models consist of the development of a tool for visualisation of the behaviour of formal models, the BRITNeY Suite [T3,C2], an application of the BRITNeY Suite visualisation tool to build a model-based prototype of a protocol facilitating communication between nodes in a mobile ad-hoc network, and a formal framework for visualisations [T5]. In the following we provide a more detailed perspective on the main contributions.

Extension of the sweep-line method to handle liveness properties

Prior to our work conducted in [T1], the sweep-line method could only check invariant properties [25, 104] and even then it was not possible in general to provide a trace from the initial state to a violating state using internal memory only, as parts of the reachability graph have been removed from memory (though work by Kristensen and Mailund existed which use external memory to provide error traces [105]).

Using our work in [T1], it becomes possible to check liveness properties, e.g., formulated using Linear Temporal Logic (LTL) [74], as well as providing error traces, using internal memory only, by storing a very compact representation of the reachability graph in internal memory. The method is shown to use significantly less memory on models with a clear notion of progress, while using only a small overhead for methods with little or no notion of progress.

Making the hash-compaction reduction technique complete

The hash-compaction technique stores the reachability graph in a highly compact manner by compressing state descriptors using a hash function. The drawback is that the hash function may not be injective, causing hash collisions, where two or more states have the same compressed state descriptor. As only one state with each compressed state descriptor is explored, this leads to parts of the reachability graph remaining unexplored. Using more than one hash function [155] the number of hash collisions can be reduced, but the basic problem, namely that the method is incomplete, persists.

The ComBack method [T2] extends the hash compaction reduction reduction technique by maintaining a spanning tree of the reachability graph rooted in the initial state. This makes it possible to resolve hash collisions on-the-fly during exploration, thereby making the method complete. The method is shown to perform reasonably well on both academic and real-life examples, trading execution time for memory usage compared to ordinary reachability graph exploration.

Development of the BRITNeY Suite visualisation tool

Prior to the development of the BRITNeY Suite [T3, C2], a lot of visualisation tools existed. Most of these tools were closed source, used a closed architecture, were tied to a single tool for formal modelling, or had more than one of these problems. Furthermore, CPN Tools [C1, 33] had no means of behavioural visualisation except using Gallash and Kristensen's COMMS/CPN library [53] for communication with external programs. This required implementing a Remote Procedure Call mechanism in each case as well as writing visualisations from scratch in a standard programming language such as Java or C++.

The BRITNeY Suite is open source and has an open architecture, which allows extension of the tool by means of plug-ins or scripts. Furthermore, it is independent of the modelling tool. This makes it possible to use the BRITNeY Suite with CPN Tools, which is also it main application. The BRITNeY Suite can also be used with other modelling tools and even other applications as well. While the BRITNeY Suite offers more than 20 plug-ins out of the box, it is easy to extend the tool to provide custom visualisations as required, due to the pluggable architecture and open source license.

Development of a model-based prototype of a protocol facilitating communication between nodes in a mobile ad-hoc network

During the B2NCW project [101] at Ericsson Denmark A/S, Telebit [47], with the resources and time available, it was deemed impossible to implement a prototype using real hardware of the protocol facilitating communication between nodes in a mobile ad-hoc network. Instead a prototype of a different, simpler, protocol was developed using real hardware, but as the extended protocol was deemed a better choice, a prototype based on a formal model was developed of this protocol.

The model-based prototype of the extended protocol was developed as a coloured Petri net model and a visualisation was developed to allow people who are not formal methods-experts to experiment with it. During the project, the BRITNeY Suite was developed and tested in a real-life setting.

Development of a formal framework for describing visualisations of the behaviour of formal models

We have devised a framework [T5], which regards both visualisations and formal models as game transitions systems, which are labelled transition systems where the transitions are separated into controllable and uncontrollable transitions. The two are executed simultaneously in a manner so that controllable transitions of the formal model are synchronised with uncontrollable transitions of the visualisation and vice versa. We require that whenever a controllable transition can be executed in the formal model or the visualisation, a corresponding uncontrollable transition can be executed in the other. The intuition is that actions initiated by the formal model (controllable transitions in the formal model) are shown to the user, and stimulation of the visualisation (controllable transitions in the visualisation) are reflected in the formal model. This approach has many advantages over previous visualisation tools. Firstly, this approach does not require changes to most formalisms, as their dynamic behaviours are usually stated using labelled transition systems as semantical domain. Secondly, it is easy to extend tools supporting visualisations in this manner as it is possible to provide a uniform interface for visualisations. Furthermore, it is difficult to forget visualisation elements as we require that the visualisation is able to accommodate any transition allowed in the formal model, so the only way to ignore a transition in the model is to do so explicitly and therefore deliberately.

4.2 Applications

This section sums up applications of the tools and methods described in this thesis. While the work of this thesis, as mentioned earlier, falls into two categories, applications of the methods are only available within the field of behavioural visualisation of formal models. The reason is that one of the verification papers, the one describing the ComBack method [T2], has only recently been published at the time of writing. The other verification method, the extended version of the sweep-line method [T1], is mainly useful for checking more complex properties, such as liveness using Linear Temporal Logic, and this does not have easy accessible tool support in tools supporting the algorithm, making real-life applications difficult. The lack of real-life applications has diminished the requirement for improvements of the algorithm. Applications of the BRITNeY Suite fall into three categories: use of the BRITNeY Suite for visualisation, use of the BRITNeY Suite for meta-visualisation, and other uses of the BRITNeY Suite. Each of these categories will be explained in the following.

Visualisation

The BRITNeY Suite has of course been used for visualisation of formal models in numerous cases. One such example is of course Kristensen, the author of this thesis, and Nørgaard's model-based prototyping of a protocol facilitating communication between nodes in mobile ad-hoc networks [T4]. Another application performed by Jørgensen and Lassen is visualisation of a formal model of blanc loan applications [94] for requirements engineering. The BRITNeY Suite has also been used by Jørgensen, Lassen, and Aalst to verify that a formal model of requirements for an electronic patient record [144] corresponds to the intended system.

Meta-visualisation

The BRITNeY Suite has also been used to implement visualisation of other formalisms by translating them into CP-nets. One such example is visualisation of UML [131] sequence diagrams, which is done independently by Machade et al. in [114] and Ribeiro and Fernandes in [145]. Both of these papers also present an industrial example of this. The BRITNeY Suite is also used to implement a workflow simulator based on coloured workflow nets [162] by Kristian Bisgaard Lassen from the University of Aarhus, Denmark. This work uses the idea of visualisations as games to develop a single visualisation, which can be used for any coloured workflow net model.

Other Applications of the BRITNeY Suite

The BRITNeY Suite has also been used in other ways. Riahi Bilel from Faculté des Sciences de Tunis (FST) uses the BRITNeY Suite to integrate algorithms written in Java with a CPN model by writing a visualisation that does not actually show anything, but only performs the required calculations. The author of this thesis has used the BRITNeY Suite to load CPN models from the command-line, which is not possible or feasible to implement using CPN Tools. György Balogh from Vanderbilt University, USA, integrate the CPN simulator into Morse et al.'s HLA (High Level Architecture) [84,128], by writing glue code as an extension of the BRITNeY Suite.

4.3 Future Work

As can be seen in Sects. 4.1 and 4.2, the goal of this thesis, namely to construct and improve methods for locating errors in computer systems, has been reached. We have focused on a tool for visualisation of the behaviour of formal models, and shown, via our own and other people's case studies, that this tool and method is indeed very useful for improving formal models. We have improved the state-of-the-art of methods for reachability graph, thereby making it possible to analyse even larger systems using this analysis approach. Still a lot of work remains, though. In Sects. 2.5.1 and 3.5.3 we provide several interesting directions for future work. The rest of this thesis is dedicated to briefly summarise this.

Improvement of methods and tools for behavioural verification using reachability graphs

The ComBack method [T2] can be extended and combined with other reduction techniques in different ways. Interesting ways to do that is to combine it with partial order reduction techniques [28, 136], which are known to reduce the in-degrees, as this would minimise the number of reconstructions required by the algorithm. It would also be interesting to combine the method with the sweep-line method, so states in front of the sweep-line are cached, also reducing the number of reconstructions.

In order for the methods described to be really useful, they should be implemented in tools which make it easy to use them on real models. Such a tool would need to implement user-friendly ways to specify properties to check. This includes user-friendly ways to specify properties that hold in a given state and natural ways to combine such properties into more complex properties stating facts about the dynamics of the formal model. We suggest looking at SPIN's

never-claims [77], which formulate properties in the same language as the formal models, Petri's Facts [139], which are Petri net transitions which must never be enabled, and Cardelli and Gordon's ambient logic [16], which state properties of the ambient calculus using a syntax that closely resembles the syntax used to specify the ambient calculus models. All of these approaches use the formalism itself or something very similar to specify properties, which is very different from CPN Tools, which uses Standard ML and temporal logics such as LTL and CTL to specify properties.

Furthermore, to support development of even better reduction techniques, a test-suite must be devised to test the methods. Such a test-suite must be able to automatically perform a large number of executions of verifications of various properties using different reduction techniques on a varied selection of models. Furthermore, it should be easy to navigate the results and track improvement over time.

Improvement of the BRITNeY Suite for behavioural visualisation

The BRITNeY Suite has already been used in numerous applications, but especially for more advanced applications improvements can be made. Firstly, the documentation is not completely satisfactory, and could be improved. Furthermore, some technical choices would be made differently today. The first choice would be to use SOAP web-services [63] instead of the currently used XML-RPC [170]. SOAP, today, enjoys wider acceptance and supports the Web Service Definition Language (WSDL) [20] for describing services, which makes it easier to integrate the BRITNeY Suite with other tools for formal modelling thanks to the wide availability of implementations of both SOAP and WSDL. As no such implementation exist for Standard ML, the implementation language of the CPN simulator used in CPN Tools [C1, 33], a complete replacement of the current implementation is unlikely to happen, however. Another, more promising, change is to use the Eclipse [41] platform rather than the custom plug-in mechanism used today. This would make it easier to develop for the BRITNeY Suite, as the Eclipse IDE could be used to debug and single-step through code, something which is not possible today. Furthermore, such a change would enable the integration of an IDE for developing (Java) code with a tool for specification using formal models and for visualising said specifications. This would make it possible to keep the specification very close to the actual implementation.

The implementation of visualisations as games, as implemented by the BRITNeY Suite at the time of writing, mainly focuses on events, which is not completely satisfactory for formalisms that are both state and event oriented, such as CP-nets. It would be nice to make synchronisation of states easy as well.

Use of behavioural visualisation to convey results of behavioural verification

The final direction for future work, we present in this thesis is a combination of the two areas dealt with in this thesis, namely behavioural verification of formal models and behavioural visualisation of formal models. The idea is to use a visualisation to present counter-examples to users. Violations of simple properties can often be visualised easily, e.g., to prove that invariant properties do not hold, we can just show an execution sequence leading to a violating state, whereas violations of other kinds of properties are not that easy to present to the user. It is quite difficult to present counter-examples to the existence of

winning strategies of games or to the validity of a CTL formula, however, as they are basically annotated reachability graphs.

The idea is to let the user contend against the computer to prove the existence of a winning strategy or the validity of the CTL formula. The user believes that the property holds, while the computer has a counter-example, proving that the property does not hold. The user selects certain transitions to execute and the computer selects other transitions. The user selects transitions using a visualisation, and the transitions selected by the computer are shown using the same visualisation. The idea is that at some point the computer will select an unanticipated transitions (or the user is unable to select an anticipated transition), which convince the user that winning strategy cannot exist/that the CTL formula does not hold. This can be because the model is wrong (if the computer performs an action not permitted by the specification or because the user cannot choose a transition which should be possible according to the specification), in which case the model needs to be changed. It is also possible that the specification is wrong, in which case the specification has to be modified and the model updated accordingly.

Part II

Papers

Chapter 5

Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method

The paper *Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method* presented in this chapter has been published as a conference paper [T1].

[T1] T. Mailund and M. Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 177–191. Springer-Verlag, 2004.

The version presented here is identical to the conference paper except for minor typographical changes.

Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method

Thomas Mailund Michael Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mailund,mw}@daimi.au.dk

Abstract

This paper is concerned with a memory-efficient representation of reachability graphs. We describe a technique that enables us to represent each reachable marking in a number of bits close to the theoretical minimum needed for explicit state enumeration. The technique maps each state vector onto a number between zero and the number of reachable states and uses the sweep-line method to delete the state vectors themselves. A prototype of the proposed technique has been implemented and experimental results are reported.

Keywords: Verification; state space methods; state space reduction; memory efficient state representation; the sweep-line method.

5.1 Introduction

A central problem in the application of reachability graph (also known as state-space) methods is the memory usage. Even relatively simple systems can have an astronomical number of reachable states, and when using basic exhaustive search [73], all states need to be represented in memory at the same time. Even methods that explore only parts of the reachability graph [6, 59, 134, 160] or explore a reduced reachability graph [46, 85, 92], often need to store thousands or millions of states.

When storing states explicitly – as opposed to using a symbolic representation such as Binary Decision Diagrams [12, 13] – the minimal number of bits needed to distinguish between N states is $\lceil \log_2 N \rceil$ bits per state. In a system with R reachable states we should therefore be able to store all reachable states using only in the order of $R \cdot \lceil \log_2 R \rceil$ bits. The number of reachable states, R, however, is usually unknown until after the reachability graph exploration; rather than knowing the number of reachable states we know the number of *syntactically possible* states S, where S is usually significantly larger than R. To distinguish between S possible states $\lceil \log_2 S \rceil$ bits are needed, so to store the R reachable states $R \cdot \lceil \log_2 S \rceil$ bits are needed. Additional memory will be needed to store transitions.

In this paper we consider mapping the state vectors of size $\lceil \log_2 S \rceil$ bits (the full state vectors or markings) to representations of length $\lceil \log_2 R \rceil$ (the condensed representations), in such a way that full state vectors can be restored when the reachability graph is subsequently analysed. Our approach is the following: We conduct a reachability graph exploration and assign to each new

unprocessed state a new number, starting from zero and incrementing with one after each assignment. The states are in this way represented by numbers in the interval $0, \ldots, R-1$. Since the state representation obtained in this way has no relation to the information stored in the full state vector, the condensed representation cannot be used to distinguish between previously processed states and new states. To get around this problem, we keep the original (full) state vectors in a table as long as needed to recognise previously seen states. The *sweep-line method* [25, 104] is used to remove the full state vectors when they are no longer needed, from memory.

In this paper we will use Place/Transition Petri nets [36] (P/T net) formalism as example to illustrate the different memory requirements needed to distinguish between the elements of the set of syntactically possible states and the set of reachable states. The use of P/T nets is only an example, the presented method applies to all formalisms where the sweep-line method can be used.

The paper is structured as follows: In Sect. 5.2 we summarise the notation and terminology for P/T nets and reachability graphs that we will use. In Sect. 5.3 we describe the condensed representation of a reachability graph, how this representation can be traversed, and how to restore enough information about the full state vectors to verify properties about the original system. In Sect. 5.4 we consider how the condensed representation can be calculated and in Sect. 5.5 we describe how the sweep-line method can be used to keep memory usage low during this construction. In Sect. 5.6 we report experimental results and in Sect. 5.7 we give our conclusions.

5.2 Petri Nets and Reachability Graphs

In this section we define *reachability graphs* of Place/Transition Petri nets.

Definition 5.1 *A **Place/Transition Petri net** is a tuple $\mathcal{N} = (P, T, F, m_I)$ where P is a set of* places*, T is a set of* transitions *such that $P \cap T = \emptyset$, $F \subseteq P \times T \cup T \times P$ is the* flow-relation*, and $m_I : P \to \mathbb{N}$ is the* initial marking*.*

We will use the usual notation for pre- and post-sets of nodes $x \in P \cup T$, i.e., $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. The state of a P/T net is given by a *marking* of the places, which is formally a multi-set over the places $m : P \to \mathbb{N}$. Since sets are a special cases of multi-sets, we will use the notation $\bullet x$ to denote both the set $\bullet x$ as defined above, but also the multi-set given by $y \mapsto 1$ when $y \in \bullet x$ and $y \mapsto 0$ when $y \notin \bullet x$. We will assume that the relations $<$, $\leq$, $>$, and $\geq$, and operations $+$ and $-$, on multi-sets are defined as usual, i.e. for two multi-sets, $m_1, m_2 : P \to \mathbb{N}$, $m_1 \leq m_2 \iff \forall p \in P.m_1(p) \leq m_2(p)$, $m_1 < m_2 \iff m_1 \leq m_2 \wedge m_1 \neq m_2$, $(m_1 + m_2)(p) = m_1(p) + m_2(p)$, and $(m_1 - m_2)(p) = m_1(p) - m_2(p)$ when $m_1 \leq m_2$ and $m_1 - m_2$ is undefined when $m_1 \not\leq m_2$.

Definition 5.2 *A transition $t \in T$ is **enabled** in marking $m : P \to \mathbb{N}$ if $m \geq \bullet t$. If t is enabled in m, it can **occur** and lead to marking m'. This is written $m\,[t\rangle\,m'$, where m' is defined by $m' = (m - \bullet t) + t\bullet$.*

We will use the common notation $m\,[\sigma\rangle\,m'$ for $\sigma = t_1 t_2 \ldots t_n \in T^*$ to mean $\exists m_i : P \to \mathbb{N}$ for $i = 0, \ldots, n$ such that $m = m_0$, $\forall i = 0, \ldots, n-1.m_i\,[t_i\rangle\,m_{i+1}$, and $m' = m_n$. We will also write $m\,[*\rangle\,m'$ to mean $\exists \sigma \in T^*$ such that $m\,[\sigma\rangle\,m'$. We say that a marking m' is *reachable* from another marking m if $m\,[*\rangle\,m'$ and we let $[m\rangle = \{m' \mid m\,[*\rangle\,m'\}$ denote the set of markings reachable from m. When we talk about the set of *reachable markings* of a P/T net, we usually mean the

set of markings reachable from the initial marking, i.e., $[m_I\rangle$. We will use R to denote the number of reachable markings, i.e., $R = |[m_I\rangle|$.

The reachability graph of a P/T net is a rooted graph that has a vertex for each reachable marking and an edge for each possible transition from one reachable marking to another.

Definition 5.3 *A **graph** is a tuple* $(V, E, \mathsf{src}, \mathsf{trg})$ *where* V *is a set of* vertices*,* E *is a set of* edges*, and* $\mathsf{src}, \mathsf{trg} : E \to V$ *are mappings assigning to each edge a* source *and a* target*, respectively. A **rooted graph** is a tuple* $(V, E, \mathsf{src}, \mathsf{trg}, r)$ *such that* $(V, E, \mathsf{src}, \mathsf{trg})$ *is a graph and* $r \in V$ *is the* root*.*

Definition 5.4 *Let* $\mathcal{N} = (P, T, F, m_I)$ *be a P/T net. The **reachability graph** of* $\mathcal{N}$ *is the rooted graph* $(V, E, \mathsf{src}, \mathsf{trg}, r)$ *defined by:*

- $V = [m_I\rangle$ *– the set of nodes is the set of reachable markings.*
- $E = \{(m, t, m') \in V \times T \times V \mid m\,[t\rangle\, m'\}$ *– the set of edges is the set of transitions from one reachable marking to another.*
- src *is given by* $\mathsf{src}(m, t, m') = m$*.*
- trg *is given by* $\mathsf{trg}(m, t, m') = m'$*.*
- $r = m_I$ *– the root is the initial marking.*

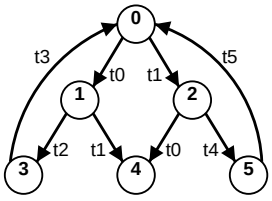
We can only represent a finite reachability graph, but the reachability graph for a P/T net need not be finite, so we put some restrictions on the P/T net we consider to ensure a finite reachability graph. The first assumption we make is that the P/T net under consideration, $\mathcal{N} = (P, T, F, m_I)$, has a finite set of places, $|P| < \infty$, and a finite set of transitions, $|T| < \infty$. The second assumption is that the net is k*-bounded* for some $k \in \mathbb{N}, k > 0$, as defined below, and consider the set of possible markings to be $\mathbb{K}^P$ where $\mathbb{K} = \{0, 1, \ldots, k\}$.

Definition 5.5 *A P/T net* (P, T, F, m_I) *is* k***-bounded** if and only if for all* $m \in [m_I\rangle$ *and for all* $p \in P$*:* $m(p) \leq k$*.*

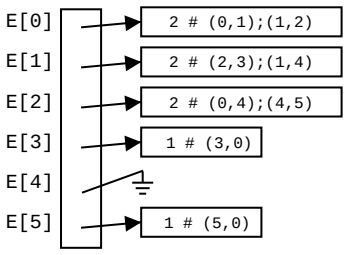
Although the assumptions above ensure that the reachability graph is finite, it is still necessary to distinguish between $|\mathbb{K}^P|$ different states when we calculate the reachability graph. If we let S denote the number of possible states, $S = |\mathbb{K}^P|$, at least $\lceil \log_2 S \rceil$ bits are needed per state. Most likely more bits will be used since the naive representation of a state vector assigns $\lceil \log_2 (k+1) \rceil$ bits per place using $|P| \cdot \lceil \log_2 (k+1) \rceil$ bits per state. Our goal is to reduce this to $\lceil \log_2 R \rceil$ bits per state.

5.3 Condensed Graph Representation

We now turn to the problem of mapping the full markings to the condensed representation. Our approach is to assign to each reachable marking a unique integer between 0 and $R - 1$, which can be represented by $\lceil \log_2 R \rceil$ bits. In this section we describe the data structure used to represent the reachability graph $\mathcal{G} = (V, E, \mathsf{src}, \mathsf{trg}, m_I)$ in this condensed form, and how to construct it from the sets V and E as calculated by the reachability graph construction algorithm. Calculating the full reachability graph and then reducing it, defeats the purpose of using a condensed representation. We only describe the algorithm in this way to present the condensed representation in an uncomplicated setting, and we will later discuss how to construct the condensed representation on-the-fly.



(a) Graph.



(b) Condensed representation.

Figure 5.1: Representation of the reachability graph. The condensed representation of the graph in (a) is shown in (b). The edge array $\mathsf{E}[\mathsf{idx}_M(v)]$ for vertex v is written in the form $n\,\#\,(\mathsf{idx}_T(t_0),\mathsf{idx}_M(v_0)\ ;\ldots;\ (\mathsf{idx}_T(t_n),\mathsf{idx}_M(v_n))$ where $n+1$ is the length of the array and the pairs represent the edges out of v. To save memory we represent a pointer to the empty array as a grounded pointer.

5.3.1 Representing the Reachability Graph

We want to represent V by the numbers 0 to $R-1$. For a marking $m \in V$ we will let $\mathsf{idx}_M(m) \in \{0,1,\ldots,R-1\}$ denote the (unique) index of m in this range. We will represent the initial marking m_I by index 0, $\mathsf{idx}_M(m_I) = 0$. With this representation of V, we can represent the set of edges as an array, E, with R entries, where each entry, $\mathsf{E}[i]$, points to an array containing the edges out of the vertex v with index i. The array pointed to by $\mathsf{E}[i]$ consists of a header – a number, indicating the length of the array, so we can later decode the array – and the edges $\{(m,t,m') \in E \mid \mathsf{idx}_M(m) = i\}$. Each edge (m,t,m') is represented as a pair $(\mathsf{idx}_T(t), \mathsf{idx}_M(m'))$ where the first element is the index of the transition – we assume some statically defined mapping $\mathsf{idx}_T : T \to \{0,\ldots,|T|-1\}$ assigning a number to each transition – and the second element is the index of the target node of the edge. An example of this representation is shown in Fig. 5.1.

Each of the pairs in the edge arrays can be represented with $\lceil \log_2 |T| \rceil + \lceil \log_2 R \rceil$ bits. In addition there is an overhead of one pointer and one number for each state in V. We assume that all edge arrays can be represented in main memory and thus that we can represent both the pointer and the number in a computer word each.[1] With this encoding, we can represent the graph $\mathcal{G} = (V, E, \mathsf{src}, \mathsf{trg}, m_I)$ using just $2wR + |E|\,(\lceil \log_2 |T| \rceil + \lceil \log_2 R \rceil)$ bits, where w denotes the number of bits in a computer word. Notice that this efficient representation is only possible because of our mapping $\mathsf{idx}_M : V \to \{0,\ldots,R-1\}$, which saves us from storing any of the R markings explicitly.

From the sets V and E of $\mathcal{G}$, the translation of the reachability graph to the condensed representation is as one would expect: We build the mapping idx_M as a table mapping nodes to numbers, allocate the array E and the individual edge arrays, and insert the data in the arrays.

5.3.2 Exploring the Condensed Reachability Graph

The condensed representation for the reachability graph explicitly contains the transition structure but does not store any information about the markings.

[1] It is possible to represent both number and pointer in $\lceil \log_2 |E| \rceil$ bits, but representing both in a computer word of a fixed size independent of $|E|$ simplifies the constructions for creating the representation on-the-fly.

```
 1: VISITED := {∅}
 2: m := m_I
 3: DFS(0)
 4:
 5: proc DFS(i) is
 6:   if i ∈ VISITED then
 7:     return
 8:   {analyse m here}
 9:   VISITED := VISITED ∪ {i}
10:   for all (t, i') in E[i] do
11:     m := m − •t + t•
12:     DFS(i')
13:     m := m + •t − t•
```

Figure 5.2: Depth-first traversal of the reachability graph. A global variable m contains the current marking during the exploration. This marking is updated before and after each recursive call. The set `visited` keeps track of the visited nodes, can efficiently be implemented as a bit vector.

For some applications, such as protocol consistency using language equivalence [9], this suffices; for other applications, however, we are interested in both marking and transition information. For such applications we need a method of recreating the markings from the transition information, without significant blowup in the memory requirements. The property that we will exploit for this is the marking equation, $m' = m - \bullet t + t\bullet$, from Def. 5.2.

When we follow an edge (i, t, i') in the condensed representation, where we know the marking of i, we calculate the marking of i' using the marking equation. If we explore the reachability graph in a depth-first manner, we can even use the rewriting of the marking equation, $m = m' - t \bullet + \bullet t$, to obtain the marking of i from the marking of i' when we return along the edge. Exploiting this, it is possible to do a depth-first graph exploration, storing only one single marking explicitly at any one time, while still having the full state vector available at each visited state. An algorithm for this is shown in Fig. 5.2.

By extending the algorithm in Fig. 5.2 with a table of sub-expressions indexed by $1, \ldots, R-1$, it can be used to check Computation Tree Logic (CTL) as in [27, Sect. 4.1], and by extending the algorithm to use nested depth-first search [74], it can be adapted to check Linear Temporal Logic (LTL).

5.4 Creating the Condensed Representation On-the-fly

To calculate the condensed representation on-the-fly we want to construct the idx_M mapping as new markings are calculated, and create the edge array at $\mathsf{E}[\,\mathsf{idx}_M(m)\,]$ as soon as the successors of m have been calculated.

A few subtleties complicate the construction: we do not know the number R, and therefore we cannot immediately allocate the array E, nor can we allocate the individual edge arrays. There is also a problem with storing the numbers in the representation of the idx_M mapping, since we do not know how many bits are needed to store the numbers $\{0, \ldots, R-1\}$. We will assume, however, that $R < 2^w$, and we can therefore represent the numbers in the table using computer words. This is potentially a waste of memory, when $\log_2 R \ll w$, but it is not likely to be a bottleneck; the majority of the memory used by the

idx_M mapping (represented as a table mapping full state vectors to numbers) will be for storing the full state vectors, which will end up using $R \cdot \lceil \log_2 S \rceil$ bits. Reduction of the memory needed for storing the full state vectors in the representation of the idx_M mapping is addressed in Sect. 5.5.

For managing the array E note that the entries in E are all of size w bits and do not depend on the total size of $[m_I\rangle$. We can work on the *entries* of E without knowing the full size of E. For handling E itself one possibility is using a dynamically extensible array [31, Chap. 18.4], expanding and relocating as needed with an amortised constant time complexity. The dynamic array approach potentially allocates an array that is too large, but will not allocate more than twice the required storage, that is, the dynamic array will use between $R \cdot w + w$ and $2 \cdot R \cdot w + w$ bits of memory (where the $+w$ is a word needed to keep track of the size of the array). To be able to relocate the dynamic array, an additional $R \cdot w$ bits of memory might be needed.

After calculating all the successors of a marking m, we can construct the edge array for m. At this point we have added all successors of m to the representation of idx_M, and since we know the number of successors, we know the size of the edge array. In the edge array we can represent each successor, m', as $\mathsf{idx}_M(m')$, using w bits. Since we have added all successors of m to the representation of idx_M, we know the maximal index, M, used in the edge array for m, so we can actually represent each successor using only $\lceil \log_2 M \rceil$ bits. With this encoding, the bits allocated per marking will now vary between the different edge arrays. To decode the arrays we must store this number with the arrays. We therefore extend the header of the edge arrays, such that it now contains both the number of edges in the array and also the number of bits allocated per marking.

5.5 Reducing Peak Memory Usage

When creating the condensed representation of the reachability graph as described in Sect. 5.4, memory is wasted because, when the algorithm terminates, the memory holds both the graph, the set of reachable markings, and the idx_M mapping. In this section we use the sweep-line method [25, 104] to keep peak memory usage small by deleting entries in the idx_M mapping.

5.5.1 The Sweep-Line Method

When constructing the reachability graph, it is neccesary to distinguish between new states and already visited states. For this we need to store the already visited states in memory. However, there is no need to store any states that are not reachable from the unprocessed states. Once a state is no longer reachable from the unprocessed states, it can be safely removed from memory.

The sweep-line method exploits this observation to delete states, using an approximation of the reachability relation, called a *progress measure*. The progress measure provides an ordering of the markings; states ordered less than the unprocessed states are assumed to be unreachable from the unprocessed states, and can therefore be deleted.

Definition 5.6 (Def. 3 in [104]) *For a P/T net (P, T, F, m_I) a **progress measure** is a tuple $\mathcal{P} = (\mathcal{V}, \sqsubseteq, \psi)$ where $\mathcal{V}$ is a set of* progress values*, $\sqsubseteq$ is a* partial order *of $\mathcal{V}$, and $\psi : \mathbb{N}^P \to \mathcal{V}$ is a mapping assigning a progress value to each marking. We say that $\mathcal{P}$ is **monotone** if $m\,[*\rangle\, m'$ implies $\psi(m) \sqsubseteq \psi(m')$.*

For monotone progress measures, the assumption that states with lower progress values are unreachable from the unprocessed states, is correct. For non-monotone progress measures, it is no longer safe just to delete states. To address this problem, we save the target nodes of edges that are not monotonic – so-called *regress edges:* (m, t, m') such that $\psi(m) \not\sqsubseteq \psi(m')$ – as *persistent* markings and never delete persistent markings. The states saved as persistent in a sweep of the state space are either previously seen states or new states; there is no way for the algorithm to know which. When we see regress edges, we therefore perform another sweep, using the new persistent states as roots for the sweep. We repeat this until we no longer find new persistent states. For details of this algorithm, see [104]. A detailed example of the construction and optimisation of a progress measure can also be found in [104].

The observation used in the sweep-line method to delete states can also be used to clean up the idx_M mapping. When constructing the condensed graph representation, we only need to store the index mapping of markings we can reach from the currently unprocessed states. Using the sweep-line method for exploring the reachability graph, we can reduce the peak memory usage by deleting states in the set V and the idx_M mapping. Deleting states is only safe if the progress measure is monotone; otherwise, the condensed graph may be an unfolding of the full graph. This is treated in Sect. 5.5.2.

The algorithm combining the sweep-line method and the construction of the condensed graph representation is shown in Fig. 5.3. Like the sweep-line algorithm, this algorithm performs a number of sweeps until it no longer finds new persistent states (lines 7–8). Each sweep (lines 10–29) consists of processing unprocessed states in order of their progress measure (lines 14–16), assigning indices to their previously unseen successors (lines 20–21), and either adding the new successors to the set of unprocessed states (line 23) or to the set of persistent states and roots for the next sweep (lines 25–26). When all successors of a state are processed, the edge array is updated (line 27) using the method CREATEEDGEARRAY (lines 31–36) as described in Sect. 5.4, and states behind the sweep-line are removed from the set V and the index mapping idx_M (lines 28–29).

By using this algorithm we only store a subset of the reachable markings explicitly while creating the condensed graph. This enables us to construct the reachability graph, in the condensed representation, in cases where storing all reachable markings in memory is impossible.

5.5.2 An Unfolding of the Reachability Graphs

When we use a non-monotone progress measure, the reachability graph obtained from the algorithm in Fig. 5.3 is not the reachability graph from Def. 5.4; rather it is an *unfolding* of this graph [118, Chap. 13]. For poor choices of progress measures, this unfolded graph can be much larger than the original reachability graph, completely eliminating the benefits of reduction. For good choices of the progress measures, the blowup in size will be manageable and the condensed representation of nodes more than compensates for the graph unfolding. It is important to consider the relationship between the unfolded graph and the original reachability graph, to know which properties are preserved by the unfolding.

The unfolding is due to regress edges – edges along which the progress measure decreases. When following a regress edge we may reach a state which has previously been explored and since the actual marking has been deleted, we do not recognise it and explore its successor states again.

```
 1: V := {m_I}
 2: Roots := {m_I}
 3: Persistent := ∅
 4: idx_M(m_I) := 0
 5: n := 1
 6:
 7: while Roots ≠ ∅ do
 8:   SWEEP(Roots, V, Persistent, idx_M, n)
 9:
10: proc SWEEP(Roots, V, Persistent, idx_M, n) is
11:   U := Roots
12:   Roots := ∅
13:   while U ≠ ∅ do
14:     select m ∈ U s.t. ∄m' ∈ U : ψ(m') ⊏ ψ(m)
15:     U := U − {m}
16:     X := {t, m' | m [t⟩ m'}
17:     for all (t, m') ∈ X do
18:       if m' ∉ V then
19:         V := V ∪ {m'}
20:         idx_M(m') := n
21:         n := n + 1
22:         if ψ(m) ⊑ ψ(m') then
23:           U := U ∪ {m'}
24:         else
25:           Persistent := Persistent ∪ {m'}
26:           Roots := Roots ∪ {m'}
27:     E[idx_M(m)] := CREATEEDGEARRAY(X, idx_M)
28:     V := {m ∈ V | ∃m' ∈ U : ψ(m') ⊑ ψ(m)} ∪ Persistent
29:     idx_M := {m ↦ i | m ∈ V ∧ idx_M(m) = i}
30:
31: proc CREATEEDGEARRAY is
32:   M := max{idx_M(m') | (t, m') ∈ X}
33:   A := allocate 2 · w + |X| · (⌈log_2 |T|⌉ + ⌈log_2 M⌉) bits
34:   A.header := (|X|, ⌈log_2 M⌉)
35:   A.edges := (idx_T(t), idx_M(m')) for each (t, m') ∈ X
36:   return A
```

Figure 5.3: The sweep-line method for obtaining a condensed graph representation.

One can easily define the unfolded graph, $\mathcal{G}^u$, and show that it is bisimilar to the full reachability graph [118, Chap. 13]. This result is especially interesting in the context of model checking, since bisimulation is known to preserve CTL* in the sense of Theorem 5.1, which in turn implies that both CTL and LTL, the most commonly used temporal logics for model checking, are preserved.

Theorem 5.1 (From [27, Chap. 12]) *If $\mathcal{G}$ and $\mathcal{G}'$ are bisimilar then for every CTL* formula ϕ we have $\mathcal{G} \models \phi \Leftrightarrow \mathcal{G}' \models \phi$.*

5.6 Experimental Results

In order to validate and evaluate the performance of the new algorithm a proof-of-concept implementation has been developed. For the theoretical presenta-

Table 5.1: Database Replication Protocol.

	Full Reachability Graph				**Sweep-Line based Algorithm**					
$\|D\|$	States	Avg	Memory	Time	States	Peak	Memory	(%)	Time	(%)
5	407	146	59,422	0	813	33	8,070	(14)	0	
6	1,460	169	246,740	0	2,919	88	26,548	(11)	1	(-)
7	5,105	191	975,055	3	10,209	251	88,777	(9)	7	(233)
8	17,498	214	3,744,572	15	34,995	738	297,912	(8)	35	(233)
9	59,051	237	13,995,087	66	118,101	2,197	993,093	(7)	155	(235)
10	196,832	259	50,979,488	286	393,663	6,572	3,276,80	()	665	(233)

tion in the previous sections we used Place/Transition Petri nets; the techniques introduced, however, generalise to higher level net classes, such as *coloured Petri nets* (CPN) [91], in a straightforward manner. The prototype is build on top of the Design/CPN tool [37], a tool for the construction and analysis of CPNs. The prototype is implemented in the *Standard ML* (SML) programming language [159] and the progress measure is provided by the user as an SML function.

Since the Design/CPN tool is used for analysing CPN models the markings of the nets are not multi-sets over places but multi-sets over more complex data types. Consequently the markings are not integer vectors of length $|P|$, but variable-length encodings of the more complex markings. On the edges of the reachability graph it is no longer sufficient to store transitions, also the bindings are needed.

The prototype implementation of the new algorithm is slightly simpler than the algorithm described in this paper. We do not implement the variable-length numbers for node indices, but represent each index as a four byte computer word. This greatly simplifies the implementation but uses slightly more memory for smaller systems and limits the prototype to models with less than 2^{32} states, which is no serious limitation.

All experiments were conducted on a 500Mhz Pentium III Linux PC with 128 Mb of RAM.

Database Replication Protocol. The first example we consider is a database replication protocol [91, Sect. 1.3]. The protocol describes the communication between a set of database managers for maintaining consistent copies of a distributed database. When a database manager updates its local copy of the database it broadcasts an update request to all other database managers who then perform the update on their local copies and then acknowledge that the update has been performed. The progress measure for the protocol is based on the control flow of the database managers and an ordering on the database managers. See [104] for details.

Table 5.1 shows the performance of full reachability graph generation compared with the new algorithm. The $|D|$ column shows the number of database managers in the different configurations, the following four columns show the values for the full reachability graph, and the last four columns show the values for the new algorithm. In the full reachability graph columns the *States* column shows the number of states for each configuration, the *Avg* column shows the average number of bytes in the state vector in the different configurations, the *Memory* column shows the total memory usage in bytes for storing all states, and the *Time* column shows the time used for calculating the reachability graph in seconds. In the sweep-line columns the *States* column shows the

Table 5.2: Stop and Wait Communication Protocol.

	Full Reachability Graph				**Sweep-Line based Algorithm**					
Packets	States	Avg	Memory	Time	States	Peak	Memory	(%)	Time	(%)
20	5,286	145	766,470	17	5,286	287	62,759	(8)	24	(141)
40	10,706	146	1,563,076	35	10,706	287	84,726	(5)	50	(143)
60	16,126	146	2,354,396	53	16,126	287	106,406	(5)	77	(145)
80	21,546	146	3,145,716	71	21,546	287	128,086	(4)	103	(145)
100	26,966	146	3,937,036	89	26,966	287	149,766	(4)	129	(145)

number of states explored by the sweep-line algorithm, the *Peak* column shows the peak number of states stored during the exploration, the *Memory* column shows the number of bytes used for storing the states in the condensed representation plus the states in *Peak*, the number in the parentheses indicates the memory consumption of the condensed representation as a percentage of the full representation, the *Time* column shows the time used for calculating the condensed graph, and the number in parentheses shows the amount of time used for calculating the condensed representation as a percentage of the amount of time used to generate the full representation.

In the database replication protocol all states but the initial state are explored twice by the sweep-line algorithm, and consequently the condensed graph has twice as many nodes as the full graph and the time for calculating the condensed graph is roughly twice as long as the time for calculating the full reachability graph. The *Memory* in the sweep-line columns is calculated as $4 \cdot States + Avg \cdot Peak$ since one computer word (4 bytes) is used for representing each condensed state and $Avg \cdot Peak$ bytes are used for representing the states on the sweep-line. We only compare the memory usage for storing the states, as the memory usage for storing the remaining graph structure would be comparable for the two methods. Although the unfolded graph generated by the sweep-line method contains twice as many nodes as the original reachability graph the memory usage – as seen in the two Memory columns – is significantly improved. For four database managers the reduction is down to around 20%, while for nine database managers the reduction is further improved, down to around 7% of the full representation.

Stop and Wait Communication Protocol. The second example is a stop-and-wait communication protocol [102]. The protocol is parameterised with the number of packets to be sent. We use the number of packets successfully received as a monotone progress measure [25]. The performance is shown in Table 5.2. Here the # *packets* column shows the number of packets to be transmitted in the different configurations; the remaining columns have the same meaning as in Table 5.1.

For this model the peak number of states fully stored in the sweep-line method does not increase for larger configurations. As the number of packets increases the total number of states increases, but the number of states with the same progress measure does not. As for the database replication protocol, the experiments shows significant memory reduction – from around 8% for 20 packets to around 4% for 100 packets – at the cost of a slight increase in runtime – an increase about 45%–50% of the runtime of the full reachability graph algorithm in all configurations.

5.7 Conclusion

In this paper we have presented a condensed representation of the reachability graph of P/T nets. The condensed graph represents each marking with a number in $\{0, 1, \ldots, R-1\}$, where $R = |[m_I\rangle|$, and avoids representing markings explicitly. We have developed an algorithm that constructs this representation exploiting local information about successor markings only to represent edges efficiently without knowing R, and dynamic arrays for storing edge information for each node. Using the sweep-line method we are able to reduce peak memory usage during the construction of the graph representation. When the progress measure used is monotone, the graph is isomorphic to the original reachability graph, and when the progress measure is non-monotone the graph is bi-similar to the original graph.

We have demonstrated the performance of the new algorithm using two examples. The chosen examples have a quite clear notion of progress, so the sweep-line method performs well, and the amount memory used to store the reduced graphs is significantly less than the amount of memory used to store the full graphs. The presented algorithm will not perform well on systems with little or no progress. An example of a system with little progress is the Dining Philosophers problem. If we use the number of eating philosophers as progress measure, we will at some time during the construction store nearly all states, and the memory used for storing the compact representation is overhead. Compared to the amount of memory used for storing the full state vectors, this amount is not significant, however, and the only real disadvantage is that we still use extra time for the construction. If the number of reachable states is close to the number of syntactically possible states, the amount of memory used for the condensed representation is comparable to the amount of memory used for the full representation, and little is gained from using the new algorithm.

By exploiting the marking equation of P/T nets, the ability to calculate the predecessor or successor of a state given a transition, we are able to reconstruct the markings of the reduced nodes while exploring the graph. In general, when the predecessors and successors can be deterministically determined, this approach can be used. If only successors can be calculated deterministically, the reachability graph can still be traversed and states reconstructed, by saving the current state on the depth-first stack before processing successors.

The algorithm presented here resembles the approach used in [61], where the basic sweep-line method (applicable to monotone progress measures only) was used to translate the reachability graph of a CPN model to a finite state automaton, which in turn was used to check language equivalence between a protocol specification and its service specification. In this approach the automaton is constructed by writing edge-information onto a disk before the sweep-line method garbage collects the edges, and this edge-information is the processed by another tool to translate it to an automaton. On the disk the states are represented as numbers, thus reducing memory consumption when the automaton is constructed from the file.

Using the graph construction algorithm presented in this paper, the potentially expensive step of going through a disk-representation can be avoided when constructing the language automaton. Furthermore, with the algorithm in Fig. 5.2 it is possible to traverse the graph reconstructing state information after the graph is constructed. The results from Sect. 5.5.2, relating the reachability graph to the unfolded graph, can also be used to generalise the method from [61] to non-monotone progress measures. In [61] the basic sweep-line method from [25] is used, guaranteeing that the automaton generated represents the language of the protocol being analysed. The results in Sect. 5.5.2

ensure that, when using non-monotone progress measures, the unfolded graph is language equivalent to the original reachability graph.

The new algorithm is designed for explicit state reachability graph analysis. For condensed state representation, such as finite automata [78], or for symbolic model checking [13,121], where states are represented as e.g., Binary Decision Diagrams [12], the memory used for storing a set of states does not depend directly on the number of states in the set, but on regularity in the state information. Deleting states during the graph construction, as the sweep-line method does, will not necessarily reduce memory usage. On the contrary, deleting states can actually increase the memory needed to store the set of states. Combining the new algorithm with symbolic model checking, therefore, does not appear to be immediately possible.

The new technique reduces the memory usage using knowledge about the number of reachable states, and complements techniques that are aimed at efficiently representing arbitrary states from the set of syntactically possible states. The state representation in SPIN [75], Design/CPN [23], and MARIA [119], for example, exploit modularity of the system being analysed to share parts of the state vector between different states. LoLA [151] exploits invariants to avoid storing information that can be derived from the invariant. Using one or more of these approaches one can represent sets of arbitrary states efficiently, though at least $\lceil \log_2 S \rceil$ bits are still needed per state to distinguish between S syntactically possible states. [57] considers storing sets of markings efficiently using very tight hash tables, which allows storing sets of states using less than $\lceil \log_2 S \rceil$ bits per state, but using the knowledge about the number of reachable states is not considered. Representing arbitrary states efficiently benefits the algorithm presented here as well, by reducing the memory needed for the table mapping states to indices. The reduction differs from probabilistic methods such as bit-state hashing [72, 76] and hash-compaction [155, 172], where all possible states are, in a sense, mapped onto a range $\{0, 1, \ldots, n\}$, for some n, but with a mapping that may not be injective on $[m_I\rangle$. The states are in this way also represented in a condensed form, but since hash collisions can occur, full coverage of the reachability graph cannot be guaranteed.

With the algorithm presented here, the sweep-line method can be used for checking more general properties than just state properties as in [104]. In particular, checking CTL* formulae, and thereby CTL and LTL formulae, now becomes possible. Future work includes using this in case studies.

Chapter 6

The ComBack Method – Extending Hash Compaction with Backtracking

The paper *The ComBack Method—Extending Hash Compaction with Backtracking* presented in this chapter has been published as a conference paper [T2].

[T2] M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *Proc. of ATPN'07*, volume 4546 of *LNCS*, pages 446–464. Springer-Verlag, 2007.

The version presented here is identical to the conference paper except for minor typographical changes.

The ComBack Method – Extending Hash Compaction with Backtracking

Michael Westergaard* Lars Michael Kristensen*†
Gerth Stølting Brodal* Lars Arge*

* Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,kris,gerth,large}@daimi.au.dk

Abstract

This paper presents the ComBack method for explicit state space exploration. The ComBack method extends the well-known hash compaction method such that full coverage of the state space is guaranteed. Each encountered state is mapped into a compressed state descriptor (hash value) as in hash compaction. The method additionally stores for each state an integer representing the identity of the state and a backedge to a predecessor state. This allows hash collisions to be resolved on-the-fly during state space exploration using backtracking to reconstruct the full state descriptors when required for comparison with newly encountered states. A prototype implementation of the ComBack method is used to evaluate the method on several example systems and compare its performance to related methods. The results show a reduction in memory usage at an acceptable cost in exploration time.

6.1 Introduction

Explicit state space exploration is one of the main approaches to verification of finite-state concurrent systems. The underlying idea is to enumerate all reachable states of the system under consideration, and it has been implemented in computer tools such as SPIN [77], Murϕ [85], CPN Tools [C1], and LoLa [150].

The main drawback of verification methods based on state space exploration is the *state explosion problem* [161], and several reduction methods have been developed to alleviate this inherent complexity problem. For explicit state space exploration these include: methods that explore only a subset of the state space directed by the verification question [134, 160]; methods that delete states from memory during state space exploration [6, 25, 60]; methods that store states in a compact manner in memory [57, 76, 93]; and methods that use external storage to store the set of visited states [156]. Another approach is symbolic model checking using, e.g., binary decision diagrams [12] or multi-valued decision diagrams [96].

Of particular interest in the context of this paper is the *hash compaction method* [155, 172], a method to reduce the amount of memory used to store

†Supported by the Carlsberg Foundation and the Danish Research Council for Technology and Production.

states. Hash compaction uses a hash function H to map each encountered state s into a fixed-sized bit-vector $H(s)$ called the *compressed state descriptor* which is stored in memory as a representation of the state. The *full state descriptor* is not stored in memory. Thus, each discovered state is represented compactly using typically 32 or 64 bits. The disadvantage of hash compaction is that two different states may be mapped to the same compressed state descriptor which implies that the hash compaction method may not explore all reachable states. The probability of *hash collisions* can be reduced by using multiple hash functions [155], but the method still cannot guarantee full coverage of the state space. If the intent of state space exploration is to find (some) errors, this is acceptable. If, however, the goal is to prove the absence of errors, discarding parts of the state space is not acceptable, meaning that hash compaction is mainly suited for error detection.

The idea of the ComBack method is to augment hash compaction such that hash collisions can be resolved during state space exploration. This is achieved by assigning a unique *state number* to each visited state and by storing, for each compressed state descriptor, a list of state numbers that have been mapped to this compressed state descriptor. This information is stored in a *state table*. Furthermore, a *backedge table* stores a *backedge* for each visited state. A backedge for a state s consists of a transition t and a state number n, such that executing transition t in the predecessor state s' with state number n leads to s. The backedges stored in the backedge table determine a spanning tree rooted in the initial state for the partial state space currently explored. The backedge table makes it possible, given the state number of a visited state s, to *backtrack* to the initial state and thereby obtain a sequence of transitions (corresponding to a path in the state space) which, when executed from the initial state, leads to s, which makes it possible to reconstruct the full state descriptor of s.

A potential hash collision is detected whenever a newly generated state s is mapped to a compressed state descriptor $H(s)$ already stored in the state table. From the compressed state descriptor and the state table we obtain the list of visited state numbers mapped to this compressed state descriptor. Using the backedge table, the full state descriptor can be reconstructed for each of these states and compared to the newly generated state s. If none of the full state descriptors for the already stored state numbers is equal to the full state descriptor of s, then s has not been visited before, and a hash collision has been detected. The state s is therefore assigned a new state number which is appended to the list of state numbers for the given compressed state descriptor, and a backedge for s is inserted into the backedge table. Otherwise, s was identical to an already visited state and no action is required.

The rest of this paper is organised as follows. Section 6.2 introduces the basic notation and presents the hash compaction algorithm. Section 6.3 introduces the ComBack method using a small example, and Sect. 6.4 formally specifies the ComBack algorithm. Section 6.5 presents several variants of the basic ComBack algorithm, and Sect. 6.6 presents a prototype implementation together with experimental results obtained on a number of example systems. Finally, in Sect. 6.7, we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the basic ideas of explicit state space exploration.

6.2 Background

The ComBack method has been developed in the context of Coloured Petri nets (CP-nets or CPNs) [91], but applies to many other modelling languages for con-

current systems such as PT-nets [143], CCS [123], and CSP [71]. We therefore formulate the ComBack method in the context of (finite) labelled transition systems to make the presentation independent of a concrete modelling language.

Definition 6.1 (Labelled Transition System) *A labelled transition system (LTS) is a tuple $\mathcal{S} = (S, T, \Delta, s_I)$, where S is a finite set of **states**, T is a finite set of **transitions**, $\Delta \subseteq S \times T \times S$ is the **transition relation**, and $s_I \in S$ is the **initial state**.*

In the rest of this paper we assume that we are given a labelled transition system $\mathcal{S} = (S, T, \Delta, s_I)$. Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. An *occurrence sequence* is an alternating sequence of states s_i and transitions t_i written $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$ and satisfying $s_i \xrightarrow{t_i} s_{i+1}$ for $1 \leq i \leq n-1$. For the presentation of the ComBack method, we initially assume that transitions are *deterministic*, i.e., if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$. This holds for transitions in, e.g., PT-nets and CP-nets. In Sect. 6.5 we show how to extend the ComBack method to modelling languages with non-deterministic transitions.

We use $\rightarrow^*$ to denote the transitive and reflexive closure of Δ, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$, $n \geq 1$, with $s = s_1$ and $s' = s_n$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$, and $\mathsf{reach}(s) = \{\, s' \in S \mid s \rightarrow^* s' \,\}$ denotes the set of states reachable from s. The *state space* of a system is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges.

The standard algorithm for explicit state space exploration relies on two data structures: a *state table* storing the states that have been discovered until now, and a *waiting set* containing the states for which successor states have not yet been calculated. The state table can be implemented as a hash table, and the waiting set can be implemented, e.g., as a stack or a fifo-queue if depth-first or breadth-first exploration is desired. The state table and the waiting set are initialised to contain the initial state and the algorithm terminates when the waiting set is empty, at which point the state table contains the reachable states.

The basic idea of the *hash compaction method* [155, 172] is to use a *hash function* H mapping from states S into the set of bit-strings of some fixed length. Instead of storing the *full state descriptor* in the state table for each visited state s, only the *compressed state descriptor* (hash value) $H(s)$ is stored. The waiting set still stores full state descriptors. Algorithm 4 gives the basic hash compaction algorithm [172]. The state table and the waiting set are initialised in lines 1–2 with the compressed and full state descriptors for the initial state s_I, respectively. The algorithm then executes a while-loop (lines 4-9) until the waiting set is empty. In each iteration of the while loop, a state s is selected and removed from the waiting set (line 5) and each of the successor states s' of s are calculated and examined (lines 6-9). If the compressed state descriptor $H(s')$ for s' is not in the state table, then s' has not been visited before, and $H(s')$ is added to the state table and s' is added to the waiting set. If the compressed state descriptor $H(s')$ for s' is already in the state table, the assumption of the hash compaction method is that s' has already been visited. The advantage of the hash compaction method is that the number of bytes stored per state is heavily reduced compared to storing the full state descriptor, which can be several hundreds of bytes for complex systems. The disadvantage is that the method cannot guarantee full coverage of the state space.

Algorithm 4 Basic Hash Compaction Algorithm

```
1: STATETABLE.INIT(); STATETABLE.INSERT(H(s_I))
2: WAITINGSET.INIT(); WAITINGSET.INSERT(s_I)
3:
4: while ¬ WAITINGSET.EMPTY() do
5:    s ← WAITINGSET.SELECT()
6:    for all t, s' such that (s, t, s') ∈ Δ do
7:       if ¬ STATETABLE.CONTAINS(H(s')) then
8:          STATETABLE.INSERT(H(s'))
9:          WAITINGSET.INSERT(s')
```

Figure 6.1 shows an example state space which will also be used when introducing the ComBack method in the next section. Figure 6.1(left) shows the full state space consisting of the states $s_1, s_2, \ldots, s_6$. The initial state is s_1. The compressed state descriptors h_1, h_2, h_3, h_4 have been written to the upper right of each state. As an example, it can be seen that the states s_3, s_5, and s_6 are mapped to the same compressed state descriptor h_3. Figure 6.1(right) shows the part of the state space explored by the hash compaction method. The hash compaction method will consider the states s_3, s_5, and s_6 to be the same state since they are mapped to the same compressed state descriptor h_3. As a result, the hash compaction method does not explore the full state space.

Several improvements have been developed for the basic hash compaction method to reduce the probability of not exploring the full state space [155]. None of these improvements guarantee full coverage of the state space. For the purpose of this paper it therefore suffices to consider the basic hash compaction algorithm.

6.3 The ComBack Method

The basic idea of the ComBack method is similar to that of the hash compaction method: instead of storing the full state descriptors, a hash function is used to calculate a compressed state descriptor. When using hash compaction, the main problem is *hash collisions*, i.e., that states with different full state descriptors (such as s_3, s_5, and s_6 in Fig. 6.1) are mapped to the same compressed state descriptor. The ComBack method addresses this problem by comparing the full state descriptors whenever a new state is generated for which the compressed state descriptor is already stored in the state table. This is, however, done without storing the full state descriptors for the states in the state table. Instead the full state descriptors of states in the state table are reconstructed

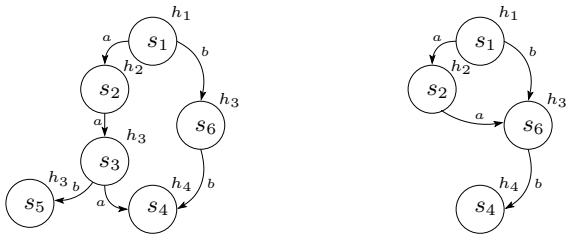


Figure 6.1: Full state space (left) and state space explored using hash compaction (right).

on-demand using *backtracking* to resolve hash collisions. The reconstruction of full state descriptors using backtracking is achieved by augmenting the hash compaction algorithm in the following ways:

1. A *state number* $N(s)$ (integer) is assigned to each visited state s.
2. The state table stores for each compressed state descriptor a *collision list* of state numbers for visited states mapped to this compressed state descriptor.
3. A *backedge table* is maintained which for each state number $N(s)$ of a visited state s stores a *backedge* consisting of a transition t and a state number $N(s')$ of a visited state s' such that $s' \xrightarrow{t} s$.

The augmented state table makes it possible, given a compressed state descriptor $H(s)$ for a newly generated state s, to obtain the state numbers for the visited states mapped to the compressed state descriptor $H(s)$. For each such state number $N(s')$ of a state s', the backedge table can be used to obtain the sequence of transitions, $t_1 t_2 \cdots t_n$, on some path (occurrence sequence) in the state space leading from the initial state s_I to s'. As we have initially assumed that transitions are deterministic, executing this occurrence sequence starting in the initial state will reconstruct the full state descriptor for s'. It is therefore possible to compare the full state descriptor of the newly generated state s to the full state descriptor of s' and thereby determine whether s has already been encountered.

Figure 6.2 (left) shows a snapshot of state space exploration using the ComBack method on the example that was introduced in Fig. 6.1. The snapshot represents the situation where the successors of the initial state s_1 have been generated, and the states s_2 and s_6 are the states currently in the waiting set. The state number assigned to each state is written inside a box to the upper left of each state. Figure 6.2 (middle) shows the contents of the state table, which for each compressed state descriptor h_i lists the state numbers mapped to h_i. Figure 6.2 (right) shows the contents of the backedge table. The backedge table gives for each state number $N(s)$ a pair $(N(s'), t)$, consisting of the state number $N(s')$ of a predecessor state s' and a transition t such that $s' \xrightarrow{t} s$. As an example, for state number 3 (which is state s_6) the backedge table specifies the pair $(1, b)$ corresponding to the edge in the state space going from state s_1 to state s_6 labelled with the transition b. For the initial state, which by convention always has state number 1, no backedge is specified since backtracking will always be stopped at the initial state.

Assume that s_2 is the next state removed from the waiting set. It has a single successor state s_3 which is mapped to the compressed state descriptor h_3 (see Fig. 6.1). A lookup in the state table shows that for the compressed state descriptor h_3 we already have a state with state number 3 stored. We therefore need to reconstruct the full state descriptor for state number 3 in

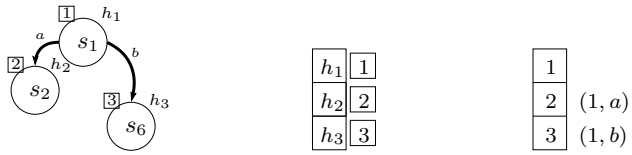


Figure 6.2: Before s_2 is processed: state space explored (left), state table (middle), and backedge table (right).

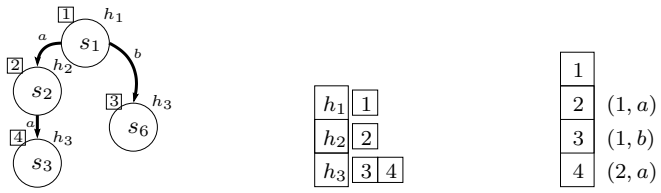


Figure 6.3: After processing s_2: state space explored (left), state table (middle), and backedge table (right).

order to determine whether s_3 is a newly discovered state. The reconstruction is done in two phases. The first phase uses the backedge table to obtain a sequence of transitions which, when executed from the initial state, leads to the state with number 3. A lookup in the backedge table for the state with state number 3 yields the pair $(1, b)$. Since 1 represents the initial state, the backtracking terminates with the transition sequence consisting of b. In the second phase, we use the transition relation Δ for the system to execute the transition b in the initial state and obtain the full state descriptor for state number 3 (which is s_6). We can now compare the full state descriptors s_3 and s_6. Since these are different, s_3 is a new state and assigned state number 4, which is added to the state table by appending it to the collision list for the compressed state descriptor h_3. In addition s_3 is added to the waiting set, and an entry $(2, a)$ is added to the backedge table for state number 4 in case we will have to reconstruct s_3 later. Figure 6.3 shows the state space explored, the state table, and the backedge table after processing s_2.

The waiting set now contains s_3 and s_6. Assume that s_3 is selected from the waiting set. The two successor states s_4 and s_5 will be generated. First, we will check whether s_4 has already been generated. As s_4 has the compressed state descriptor h_4, which has no state numbers in its collision list, it is new, and it is assigned state number 5, and an entry $(4, a)$ is added to the backedge table. Then we check if s_5 is new. State s_5 has the compressed state descriptor h_3 and a lookup in the state table yields the collision list consisting of states number 3 and 4. Using the backedge table, we obtain the two corresponding transition sequences: $(1, b)$ and $(2, a)(1, a)$. Executing the occurrence sequences: $s_1 \xrightarrow{b} s_6$ and $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3$ yields the full state descriptors for s_3 and s_6. By comparison with the full state descriptor for s_5 it is concluded that s_5 is new and the state table, the waiting set, and the backedge table are updated accordingly.

When state s_3 has been processed, the waiting set contains the states s_4, s_5, and s_6. The processing of s_4 and s_5 does not result in any new states as these two states do not have successor states. Consider the processing of s_6. We will tentatively denote the full state descriptor for the successor of s_6 corresponding to s_4 by s' as the algorithm has not yet determined that it is equal to s_4. State s' has the compressed state descriptor h_4 and a lookup in the state table shows that we have a single state with number 5 stored for h_4. The backedge table is then used starting from state number 5 to obtain the backedges $(4, a)$, $(2, a)$, and $(1, a)$. Executing the corresponding occurrence sequence $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$ yields full state descriptor for s_4, and we conclude that this full state descriptor is equal to s', so s' has already been visited and no changes are required to the state table, the waiting set or the backedge table.

Figure 6.4 shows the situation after state s_6 has been processed. The thick edges correspond to the backedges stored in the backedge table. It can be seen that the backedges stored in the backedge table determine a spanning tree rooted in the initial state in all stages of the construction (Figs. 6.2–6.4).

6.4 The ComBack Algorithm

The ComBack algorithm introduced in the previous section is listed in Algorithm 5. The first part of the algorithm (lines 1–4) initialises the global data structures. The global variable m is used to enumerate the states, i.e., assign state numbers to states, and is initially 1 since the initial state is the first state considered. The state table has an INSERT operation which takes a compressed state descriptor and a state number and appends the state number to the collision list for the compressed state descriptor. The waiting set stores pairs consisting of a full state descriptor and its number. The state number is needed when creating the backedge for a newly discovered state. The backedge table stores pairs consisting of a state number and a transition label. The empty backedge denoted $\perp$ is initially inserted in the backedge table for state number 1 (the initial state).

The algorithm then executes a while-loop (lines 6–13) until the waiting set is empty. In each iteration of the while-loop, a pair, (s, n'), consisting of a state and its state number is selected from the waiting set (line 7) and each of the successor states, s', of s is examined (lines 8–13). Whether a successor state, s', is a newly discovered state is determined using the CONTAINS procedure, which will be explained below. If s' is a newly discovered state, m is incremented by one to obtain the state number assigned to s', the state number for s' is appended to the collision list associated with the compressed state descriptor $H(s')$, and (n', t) is inserted as a backedge in the backedge table for the state s' which has been given state number m.

The procedure CONTAINS (lines 15–19) is used to determine whether a newly generated state s' has been visited before. The procedure looks up the collision list for the compressed state descriptor $H(s')$ for s', and for each state number, n, in the collision list it checks if s' corresponds to n using the MATCHES procedure. If a reconstructed state descriptor is identical to s', then s' has already been visited and tt (true) is returned. Otherwise ff (false) is returned. The procedure MATCHES (lines 21–22) reconstructs the full state descriptor corresponding to n using RECONSTRUCT procedure and returns whether it is equal to s'.

The procedure RECONSTRUCT recursively backtracks using the backedge table to reconstruct the full state descriptor for state number n. The function recursively finds the state number of a predecessor using the backedge table and calculates the full state descriptor using the EXECUTE procedure. The procedure exploits the convention that the initial state has number 1 to determine when to stop the recursion. The EXECUTE procedure (not shown) uses the transition relation Δ to compute the state resulting from an occurrence of the transition t in the state s, i.e., if $(s, t, s') \in \Delta$ then $\text{EXECUTE}(s, t) = s'$. This

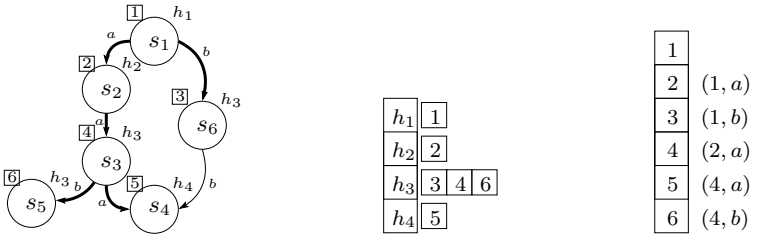


Figure 6.4: After processing s_6: state space explored (left), state table (middle), and backedge table (right).

Algorithm 5 The ComBack Algorithm

```
 1: m ← 1
 2: StateTable.Init(); StateTable.Insert(H(s_I), 1))
 3: WaitingSet.Init(); WaitingSet.Insert(s_I, 1)
 4: BackEdgeTable.Init(); BackEdgeTable.Insert(1, ⊥)
 5:
 6: while ¬ WaitingSet.Empty() do
 7:   (s, n') ← WaitingSet.Select()
 8:   for all t, s' such that (s, t, s') ∈ Δ do
 9:     if ¬ Contains(s') then
10:       m ← m + 1
11:       StateTable.Insert(H(s'), m)
12:       WaitingSet.Insert(s', m)
13:       BackEdgeTable.Insert(m, (n', t))
14:
15: proc Contains(s') is
16:   for all n ∈ StateTable.Lookup(H(s')) do
17:     if Matches(n, s') then
18:       return tt
19:   return ff
20:
21: proc Matches(n, s') is
22:   return s' = Reconstruct(n)
23:
24: proc Reconstruct(n) is
25:   if n = 1 then
26:     return s_I
27:   else
28:     (n', t) ← BackEdgeTable.Lookup(n)
29:     s ← Reconstruct(n')
30:     return Execute(s, t)
```

is well-defined since we have assumed that transitions are deterministic.

It can be seen that the ComBack algorithm is very similar to the standard algorithm for state space exploration. The main difference is that determining whether a state has already been visited relies on the CONTAINS procedure which uses the backedge table to reconstruct the full state descriptors before the comparison with a newly generated state is done. Since the backedge table at any time during state exploration determines a spanning tree rooted in the initial state for the currently explored part of the state space, we can reconstruct the full state descriptor for any visited state. It follows that the ComBack algorithm terminates after having explored all reachable states exactly once.

6.4.1 Space Usage.

The ComBack algorithm explores the full state space at the expense of using more memory per state than hash compaction and by using time on reconstruction of full state descriptors. We will now discuss these two issues in more detail. First we consider memory usage. Let w_N denote the number of bits used to represent a state number, and let w_H denote the number of bits in a compressed state descriptor. Let $|h_i|$ denote the number of reachable states mapped to the compressed state descriptor h_i. The entry corresponding to h_i

in the state table can be stored as a pair consisting of the compressed state descriptor and a counter of size w_c specifying the length of an array of state numbers (the collision list). The total amount of memory used to store the states whose compressed state descriptor is h_i is therefore given by $w_H + w_c + |h_i| \cdot w_N$. Considering all compressed state descriptors, the worst-case memory usage occurs if all collision lists have length 1. This means that the worst-case memory usage for the state table is:

$$|\mathsf{reach}(s_I)| \cdot (w_H + w_c + w_N)$$

We need at least $w_N = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing unique numbers for each state and $w_c = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing the number of states in each collision list. The worst-case memory usage for the elements in the state table is therefore:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 2 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil)$$

Consider now the backedge table. The entries can be implemented as an array where entry i specifies the backedge associated with state number i. If we enumerate all transitions, each transition in a backedge can be represented using $\lceil \log_2 |T| \rceil$ bits. Each state number in a backedge can be represented using $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits. Observing that each reachable state will have one entry in the backedge table upon termination this implies that the memory used for the elements in the backedge table is given by:

$$|\mathsf{reach}(s_I)| \cdot (\lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

The above means that the total amount memory used for the elements in the state table and the backedge table is in worst-case given by:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

This is $3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil$ bits more per visited state than the hash compaction method. The ComBack method and the hash compaction method both store the full state descriptor for those states that are in the waiting set, but the ComBack method additionally stores the state number of each state in the waiting set which implies that the ComBack method uses $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ more bits per state in the waiting set. In reality, we will not know $|\mathsf{reach}(s_I)|$ in advance, and we will therefore use a machine word (w bits) for storing state numbers. If we furthermore assume that we store each transition using a machine word and use a hash function generating compressed state descriptors of size $w_H = w$, we use a total of $5 \cdot w$ bits or 5 machine words per state, corresponding to 20 bytes on a 32-bit architecture.

6.4.2 Time Analysis.

Let us now consider the additional time used by the ComBack algorithm for reconstruction of full state descriptors. Let $\hat{h}_i = \{s_1, s_2, \ldots, s_n\}$ denote the states that are mapped to given compressed state descriptor h_i and assume that they are discovered in this order. The first state s_1 mapped to h_i will not result in a state reconstruction, but when state s_j is discovered the first time it will cause

a reconstruction of the states $s_1, s_2, \ldots s_{j-1}$. This means that the number of reconstructions caused by the first discovery of each of the states is given by:

$$\sum_{j=1}^{|\hat{h}_i|}(j-1) = \frac{|\hat{h}_i| \cdot (|\hat{h}_i|-1)}{2}$$

Any additional input edge of an already discovered state mapped to h_i will in worst-case cause all other discovered states to be regenerated. In the worst case, the additional input edges are discovered after all $|\hat{h}_i|$ states have been discovered for the first time. Let $\mathsf{in}(s)$ denote the number of input edges for a state s. The number of reconstructions caused by additional input edges is then given by:

$$|\hat{h_i}| \cdot \sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1)$$

This means that the total number of state reconstructions for a given compressed state descriptor h_i is given by:

$$\begin{aligned}
\frac{|\hat{h}_i| \cdot (|\hat{h}_i|-1)}{2} + |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1) &= \frac{1}{2}|\hat{h}_i|^2 - \frac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}(s_j) - |\hat{h}_i|^2 \\
&= -\frac{1}{2}|\hat{h}_i|^2 - \frac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\
&\leq |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j)
\end{aligned}$$

Let $\hat{H} = \{\, H(s) \mid s \in \mathsf{reach}(s_I) \,\}$ denote the set of compressed state descriptors for the set of reachable states. The number of reconstructions used for the entire state space exploration can be then be approximated by:

$$\begin{aligned}
\sum_{h_i \in \hat{H}} |\hat{h}_i| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) &\leq \sum_{h_i \in \hat{H}} \left(\max_{h_k \in \hat{H}} |h_k| \cdot \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \right) \\
&= \max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{h_i \in \hat{H}} \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\
&= \max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \mathsf{reach}(s_I)} \mathsf{in}\,(s)
\end{aligned}$$

If we assume that we are using a good hash function for computing the compressed state descriptors, then $|\hat{h}_i|$ will in practice be small (at most 2 or 3). This means that the total number of state reconstructions will be close to the sum of the in-degrees of all reachable states which is equal to number of edges in the full state space. A poor hash function will cause many state reconstructions which in turn will seriously affect the run-time performance of the algorithm. In Sect. 6.6 we will show how to obtain a good hash function in the context of CP-nets. If the backedge table is implemented as an array, we get a constant look-up time, and a state can be reconstructed in time proportional to the length of the path.

The above is summarised in the following theorem where $\{0,1\}^{w_H}$ denotes the set of bit strings of length w_H.

Theorem 6.1 *Let $\mathcal{S} = (S, T, \Delta, s_I)$ be a labelled transition system and $H : S \to \{0,1\}^{w_H}$ be a hash function. The ComBack algorithm in Algorithm 5 terminates after having explored all reachable states of $\mathcal{S}$ exactly once. The elements in the state table and the backedge table can be represented using:*

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil) \textit{ bits}$$

The total number of state reconstructions during exploration is bounded by:

$$\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \mathsf{reach}(s_I)} \mathit{in}\,(s)$$

6.5 Variants and Extensions

In this section, we sketch several variants of the basic ComBack algorithm. Variants 1 and 2 are aimed at reducing time usage while Variants 3 and 4 are aimed at reducing memory usage. Variant 5 shows how the ComBack method can be used for modelling languages with non-deterministic transitions.

Variant 1: Path Optimisation

The amount of time used on reconstruction of a state s is proportional to the length of the occurrence sequence leading to s stored in the backedge table. If the state space is constructed in a breadth-first order, the backedge table automatically contains the shortest occurrence sequences for reconstruction of states. This is not the case, e.g., when using depth-first exploration. When the state space is not explored breadth-first, it is therefore preferable to keep the occurrence sequences in the backedge table short. As an example consider Fig. 6.4. The occurrence sequences stored in the backedge table for s_4 (state number 5) is $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$, which is of length 3. A shorter path $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ has however been found when s_4 was re-discovered from s_6. When re-discovering s_4 from s_6, it is therefore beneficial to replace the backedge $(4, a)$ stored for s_4 to $(3, b)$ such that the shorter occurrence sequence $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ is stored in the backedge table. It is easy to modify the algorithm to make such simple path optimisations by storing the *depth* of each state in the waiting set along with the full state descriptor and state number. The depth of a state s stored in the waiting set is the length of the occurrence sequence through which s was explored. Whenever a state s is removed from the waiting set in line 7 of Algorithm 5, we obtain the depth d of s. By incrementing d by one, we obtain the depth of each successor state s' of s. If the RECONSTRUCT procedure (see lines 24–30 in Algorithm 5) reconstructs s' based on the backedge table using an occurrence sequences of length greater than $d+1$, then the backedge stored for s' should be changed to point to the state number of s since going via s results in a shorter occurrence sequence. It is easy to see that the above path optimisation shortens the occurrence sequences stored in the backedge table, but it does not necessary yield the shortest occurrence sequences.

Variant 2: Caching of Full State Descriptors

Another possibility of reducing the time spent on state reconstruction is to maintain a small cache of some full state descriptors for the visited states. As an example, consider Fig. 6.4 and assume that we have cached state s_3 (with state number 4) during exploration. Then we would not need to do backtracking

for state number 4 when we generate state s_5 – we can immediately see that even though states s_3 and s_5 both have the compressed state descriptor h_3, the cached full state descriptor for s_3 is not the same as the full state descriptor for s_5. Caching s_3 also yields an optimisation when we generate state s_4 (with state number 5) when processing s_6. In this case we would not have to backtrack all the way back to the initial state, but as soon as we encounter state number 4 in the backtracking process we can obtain the full state descriptor for s_3 (since it is cached), and it suffices to execute the occurrence sequence $s_3 \xrightarrow{a} s_4$ to reconstruct the full state descriptor for s_4. This shows that caching also optimises state reconstruction for non-cached states. Another way to further optimise backtracking is to re-order the states in the collision lists according to some heuristics that attempt to predict which state is most likely to be re-visited. A simple heuristic is to move a state number to the front of the collision list every time we re-encounter it.

Variant 3: Backwards State Reconstruction

Some modelling languages, including PT-nets and CP-nets, allow transitions to be executed backwards, i.e. we can obtain a function Δ^{-1} such that $\Delta^{-1}(s', t) = s \iff (s, t, s') \in \Delta$. This can be used to execute occurrence sequences from the backedge table backwards, starting from the full state descriptor of a newly generated state s', in order to determine whether s' has already been visited. This has two benefits. Firstly, we do not need to store the occurrence sequence obtained from the backedge table in memory, but can just iteratively look up a backedge in the backedge table and transform the current state using Δ^{-1}. Secondly, the backtracking process may stop early if we encounter an *invalid state*. What qualifies as an invalid state depends on the modelling formalism. A simple implementation for PT-nets and CP-nets is to consider states to be invalid if there is a negative amount of tokens on a place (which may happen when transitions are executed backwards).

Variant 4: Reconstruction of Waiting Set States

In the basic ComBack algorithm we store the full state descriptors for the states in the waiting set. This may take up a considerable amount of memory. It can be observed that we do not actually need to store the full state descriptor for states in the waiting set. It suffices to store the state number as the full state descriptor can be reconstructed from the state number and the backedge table when the state number is selected from the waiting set. This reduces memory usage at the expense of having to make up to $|\mathsf{reach}(s_I)|$ extra reconstructions of states. We can alleviate this, however, if we do depth-first exploration and cache at least the last state that was processed.

Variant 5: Non-deterministic Transition

For modelling languages with non-deterministic transitions we may have $(s, t, s') \in \Delta \wedge (s, t, s'') \in \Delta$ such that $s' \neq s''$. This means that we may not have a single unique state when executing occurrence sequences obtained from the backedge table, and a state reconstruction procedure is required that operates on sets of states. Consider the reconstruction of a visited state with number n. From the backedge table we obtain (as before) a sequence of backedges $(n_m, t_m) \cdots (n_i, t_i) \cdots (n_2, t_2)(n_1, t_1)$ where $n_1 = 1$ (the initial state). In the i'th step of the reconstruction process when considering the backedge (n_i, t_i), now have a set of states S_1 containing the states that can be reached by executing

the transition sequence $t_1 t_2 t_{i-1}$ starting in the initial state. From this set we obtain a new set of states S_2 which is the set of states obtained by executing t_i in those states of S_1 where t_i is enabled. To reduce the size of the set S_2 we observe that S_2 should only contain those states that has the same compressed state descriptor as state number n_{i+1}. The compressed state descriptor for state number n_{i+1} can be obtained from the state table. With a good hash function H, this is expected to keep the size of the sets of states considered during state reconstruction small.

Revised MATCHES and RECONSTRUCT procedures for Variant 5 are shown in Algorithm 6. The RECONSTRUCT procedure is changed to return a set of possible states matching the state number n, so MATCHES is changed to check if s is among those (line 2). The only state corresponding to state number 1 is the initial state (line 6). In line 8 we look up the number of a predecessor state in the backedge table and recursively reconstruct all states that can match that state (line 9). Then we calculate all possible successors of those states (line 10). After that we check that the state number we are looking for, n, is actually in the collision list of the compressed state descriptor of all calculated successors (line 11), and finally return the result. The algorithm will work without the weeding of states in line 11, but at the expense of considering larger state sets.

Algorithm 6 MATCHES and RECONSTRUCT procedures for Variant 5

```
 1: proc MATCHES(n, s) is
 2:   return s ∈ RECONSTRUCT(n)
 3:
 4: proc RECONSTRUCT(n) is
 5:   if n = 1 then
 6:     return {s_I}
 7:   else
 8:     (n', t) ← BACKEDGETABLE.LOOKUP(n)
 9:     S_1 ← RECONSTRUCT(n')
10:     S_2 ← {s_2 ∈ S | ∃ s_1 ∈ S_1 : (s_1, t, s_2) ∈ Δ}
11:     S_3 ← {s_2 ∈ S_2 | n ∈ STATETABLE.LOOKUP(H(s_2))}
12:     return S_3
```

6.6 Experimental Results

A prototype of the basic algorithm as described in Sects. 6.3 and 6.4 has been implemented in CPN Tools [C1] which supports construction and analysis of CPN models [91]. The algorithm is implemented in Standard ML of New Jersey (SML/NJ) [159] version 110.60.

The STATETABLE is implemented as a hash mapping (using lists for handling collisions) and the BACKEDGETABLE is implemented as a dynamic extensible array. This ensures that we can make lookups and insertions in (at least amortized) constant time. The collision list is implemented using SML/NJ's built-in list data type, which is a linked list (rather than an array with a length). A more efficient implementation of the STATETABLE could be obtained using very tight hashing [57]. This would allow us to remove some redundant bits from the compressed state descriptor. We have implemented both depth-first exploration (DFS) and breadth-first exploration (BFS).

The compressed state descriptors calculated by the hash function as well as the state numbers are 31-bit unsigned integers as SML/NJ uses the 32^{nd} bit for garbage collection. The hash function used is defined inductively on the state

of the CPN model. In CP-nets, a state of the system is a *marking* of a set of *places*. Each marking is a *multi-set* over a given *type*. We use a standard hash function for each type. We extend this hash function to multi-sets by using a *combinator function*, which takes two hash values and returns a new hash value. We extend the hash functions on markings of places to a hash function of the entire model by using the combinator function on the place hash functions.

We also implemented caching of full state descriptors as explained in Sect. 6.5. The caching strategy used is simple: we use a hash mapping from state numbers to full state descriptors, which does not account for collisions of hash values. That way, if we allocate a hash mapping of, say, size 1000, we can store at most 1000 full state descriptors in the cache. We have not implemented re-ordering of states in the collision lists, as the collision lists have length at most 2 (with two exceptions) for all our examples.

We use a test-suite consisting of three kinds of models: small examples, medium-sized examples and real-life applications. In the first category, we have three models: a model of the dining philosophers system (DP), a model of replicating database managers (DB), and a model of a stop-and-wait network protocol (SW). In the second category, we have a model of a telephone system (TS). In the last category, we have a model of a protocol (ERDP) for distributing network prefixes to gateways in a network consisting of standard wired networks and wireless mobile ad-hoc networks [103]. All of the models are parametrised: DP by the number of philosophers, DB by the number of database managers, SW by the number of packets transmitted and the capacity of the network, TS by the number of telephones, and ERDP by the number of available prefixes and the capacity of the network. We will denote each model by its name and its parameter(s), e.g. DP22 denotes DP with 22 philosophers and ERDP6,2 denotes the ERDP protocol with six prefixes and a network capacity of two.

We have evaluated the performance of the ComBack method without cache, denoted by ComBack, and with cache of size n, denoted ComBack n. We have compared the ComBack method with implementations of basic hash compaction [172], bit-state hashing [76] by means of double hashing [38] which uses a linear combination of two hash functions to compute, in this case, 15 compressed state descriptors. Instead of storing the compressed state descriptors, like hash compaction, bit-state hashing uses the values to set bits in a bit-array. Finally, we compare the ComBack method to standard state space exploration of the full state space using a hash table for storing the full state descriptors. For each model, we have measured how much memory and how much CPU time was used to conduct the state space exploration. Memory is measured by performing a full garbage collection and measuring the size of the heap. This is done every 0.5 second or 40 states, whichever comes last. As garbage collection takes time, the CPU time used is measured independently. We have measured the time three times and used the average as the result.

Table 6.1 shows the results of the experiments. For each model (column 1) and each exploration method (column 2), we show the number of nodes (states) and arcs explored (columns 3 and 4). We also show the CPU time spent (in seconds) and the amount of space (memory) used (in mega-bytes) for a depth-first traversal (DFS) and a breadth-first traversal (BFS) of the state space (columns 5, 7, 10, and 12). In addition we show how much time and memory is used relative to traversal using a standard exploration using DFS (columns 6, 8, 11, and 13) and how much memory (in bytes) is used per state (columns 9 and 14).

We note that for each model, independent of the reduction technique, either DFS performs better memory-wise than BFS or vice versa. For the more realistic examples, TS and ERDP, DFS is slower than BFS. This is due to the

Table 6.1: Experimental results

model	method	nodes	arcs	DFS					BFS				
				time		space			time		space		
				sec	%	Mb	%	/state	sec	%	Mb	%	/state
DP22	ComBack	39604	481625	2791	10337	23.0	97	608	59	219	9.8	42	260
	ComBack 100	39604	481625	800	2963	23.0	98	610	56	207	9.9	42	261
	ComBack 1000	39604	481625	98	363	23.6	100	625	57	211	10.6	45	281
	Hash compaction	39603	481609	25	93	20.8	88	550	26	96	8.4	35	222
	Bit-state	39604	481609	28	104	32.0	135	846	29	107	20.0	85	531
	Standard	39604	481625	27	100	23.6	100	625	27	100	14.3	61	380
DB9	ComBack	59051	314947	60	214	4.5	10	80	63	225	11.9	28	212
	ComBack 100	59051	314947	50	178	4.7	11	83	51	182	12.1	28	214
	ComBack 1000	59051	314947	48	171	5.5	13	98	44	157	12.9	30	229
	Hash compaction	59049	314937	25	89	1.4	3	25	27	96	10.1	23	179
	Bit-state	59051	314947	29	104	12.3	28	218	33	118	21.3	49	379
	Standard	59051	314947	28	100	43.3	100	769	28	100	43.4	100	770
DB10	ComBack	196832	1181001	286	44	15.4	9	82	307	48	43.1	25	230
	ComBack 100	196832	1181001	247	38	15.6	9	83	264	41	43.3	25	231
	ComBack 1000	196832	1181001	240	37	16.6	10	89	250	39	44.4	26	236
	Hash compaction	196798	1180790	118	18	4.9	3	26	133	21	36.8	21	196
	Bit-state	196832	1181001	138	21	12.3	7	66	152	24	46.3	27	247
	Standard	196832	1181001	643	100	174.0	100	927	693	106	174.0	100	927
SW7,4	ComBack	215196	1242386	115	319	17.5	41	85	115	319	20.1	47	98
	ComBack 100	215196	1242386	68	189	17.6	41	86	100	278	20.2	47	98
	ComBack 1000	215196	1242386	64	178	17.9	42	87	93	258	20.6	48	100
	Hash compaction	214569	1238803	33	92	5.2	12	25	37	103	9.8	23	48
	Bit-state	215196	1242386	41	114	12.3	28	60	46	128	18.3	43	89
	Standard	215196	1242386	36	100	43.0	100	210	40	100	43.1	100	210
TS5	ComBack	107648	1017490	3302	6115	51.4	84	500	103	191	17.6	29	172
	ComBack 100	107648	1017490	933	1728	51.4	83	501	102	189	17.7	29	172
	ComBack 1000	107648	1017490	207	383	51.9	85	506	107	198	18.4	30	180
	Hash compaction	107647	1017474	50	93	45.7	75	445	52	96	14.7	24	143
	Bit-state	107648	1017490	58	107	55.4	90	540	62	115	25.8	42	251
	Standard	107648	1017490	54	100	61.2	100	596	57	106	45.0	73	438
ERDP6,2	ComBack	207003	1199703	986	865	29.1	33	147	867	761	35.7	41	181
	ComBack 100	207003	1199703	259	227	29.0	33	147	481	422	35.8	41	181
	ComBack 1000	207003	1199703	205	180	29.6	34	150	402	353	36.4	42	184
	Hash compaction	206921	1199200	106	93	5.1	6	26	114	100	18.6	21	94
	Bit-state	207003	1199703	123	108	12.3	14	62	135	118	27.3	31	138
	Standard	207003	1199703	114	100	87.4	100	443	131	115	88.5	101	449
ERDP6,3	ComBack	4277126	31021101	42711	-	572.3	-	140	65354	-	708.1	-	174
	ComBack 100	4277126	31021101	18043	-	571.2	-	140	-	-	-	-	-
	ComBack 1000	4277126	31021101	23084	-	571.7	-	140	-	-	-	-	-
	Hash compaction	4270926	30975030	3341	-	113.5	-	28	20512	-	403.6	-	99
	Bit-state	4277125	31021091	3732	-	12.1	-	3	17481	-	347.9	-	85
	Standard	-	-	-	-	-	-	-	-	-	-	-	-

fact that the models resemble real systems and have more complex behaviour, which leads to very long occurrence sequences in the backedge table, and thus impacts the performance of the ComBack method. If we instrument the ComBack method with even a small cache when using DFS, or if we use BFS, processing is much faster for realistic examples. We see that the ComBack method uses quite a bit more memory than the 5 machine words predicted in the previous section. One cause for this is that the calculation in Sect. 6.4 did not take the WAITINGSET into account and only considered the *elements* of the state table and the backedge table, not the tables themselves. Furthermore, SML is not very memory-efficient, doubling usage. We also note that the standard exploration as well as the ComBack method using BFS were not able to complete due to lack of memory for the ERDP6,3 model. The hash compaction bit-state hashing were also not able to explore all states for this example (as can be seen in the nodes column of Table 6.1). This means we are comparing methods guaranteeing full coverage with methods that do not, so while the hash compaction and bit-state hashing methods seem to perform well, they do so at a cost.

Figure 6.5 shows charts depicting memory and time usage relative to standard DFS exploration (i.e. one chart for columns 5 and 7 and another chart for columns 10 and 12). These charts allow us to better understand how the different exploration methods perform compared to each other, independent of the example. We see that the values fall into 7 rectangles corresponding to 6 different exploration methods and an abnormal experiment. Rectangle 1: standard exploration; all results are near 100% on both axes, showing that when we store full state descriptors in a hash table, it does not matter whether we use DFS or BFS. Rectangle 2: hash compaction; all are near 100% on the time axis and between 2% and 100% (DFS) or 20% and 40% (BFS) on the memory axis, showing that hash compaction uses as much time as storing the full state descriptors, but significantly less space. Rectangle 3: bit-state hashing; all are near 100% time-wise, but slightly higher than 1 and 2 (this is probably because we have to calculate two hash values instead of just one). All range between 15% and 150% memory-wise. The bit-state hashing method consistently uses 12.5 mega-bytes plus the size of the waiting set, so it performs well memory-wise on models with large state spaces, but performs poorly on models with small state spaces. This means that memory optimisations are possible, but customisation is required by the user. All the show models are reasonable large, leading to reasonable performance of bit-state hashing. Rectangle 4: ComBack without cache; all are above 150% time-wise and between 10% and 100% (DFS) or 25% and 40% (BFS) memory-wise. Time is also better bounded in the BFS results. This indicates that ComBack without cache yields a reduction (it is never above 100%), and when using BFS we have better control of the time and memory used. DFS makes it possible to save more memory, but can be very costly time-wise, and sometimes we do not save any memory at all (e.g. in the Dining philosophers example, where we can end up with most of the state space in the waiting set). Rectangles 5+6: ComBack with cache; these use slightly more memory but less time than 4, in particular in the DFS case. More cache yields more memory and less time used, but the differences are not that large, and even a small cache yields great optimizations in time compared to the ComBack method with no cache at all. Rectangle 7: DB10; these points fall outside of all the other boxes. Inspection of the data in Table 6.1 shows that the DB10 example has irregular behaviour, as exploration using the standard exploration is slow as a full state descriptor for this model is large, and thus the SML/NJ garbage collector is invoked often. This yields a performance penalty and causes all other experiments, as they are relative to the standard exploration, to fall outside the other boxes. ERDP6,3 is not shown as the standard exploration was unable to terminate.

All of the shown experiments have been performed using a hash-function generating 31-bit compressed state descriptors. We have also tested the method using a hash function generating 62-bit compressed state descriptors, but have not shown those results, as the time usage is the same but more memory is consumed, as the 31-bit hash function causes few collisions. We have verified the quality of the hash-function by calculating the lengths of the collision lists for all examples. The worst case is example SW7,4, where there are 214009 collision lists of length 1, 592 lists of length 2 and 1 list of length 3, so 99.7 % of the collision lists have the minimum length. It also means that hash compaction misses at least $1 \cdot 592 + 2 \cdot 1 = 594$ states due to hash collisions.

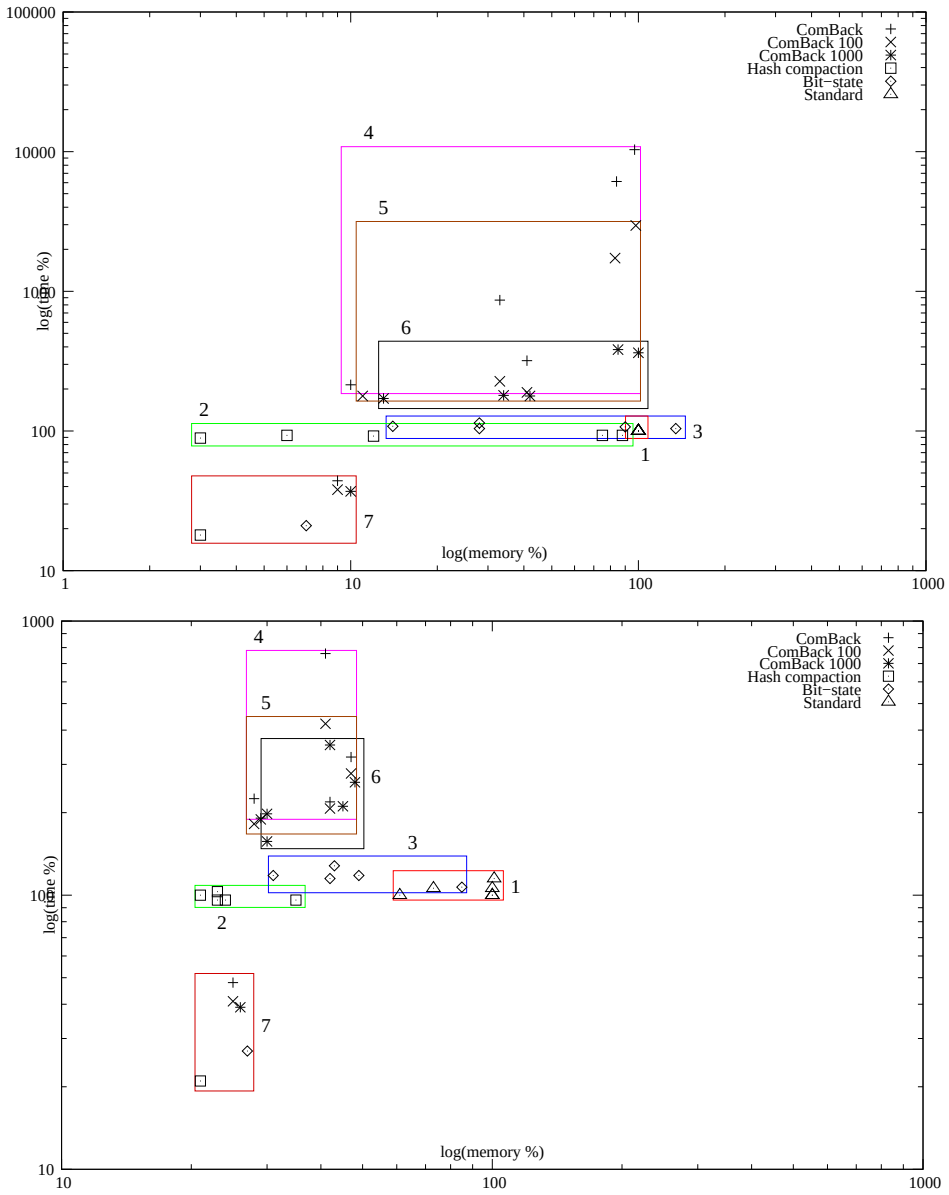


Figure 6.5: Time and memory usage for the various reduction techniques using DFS exploration (top) and BFS exploration (bottom). Values are relative to corresponding values for standard depth-first exploration.

6.7 Conclusions and Future Work

In this paper we have presented the ComBack method for alleviating the state explosion problem. The basic idea of the method is to augment the hash compaction method with a backedge table that makes it possible to reconstruct full state descriptors and ensure full coverage of the state space. We have made a prototype implementation of the method in CPN Tools and our experimental results demonstrate that the method (as expected) uses more time and memory than hash compaction, but less memory than ordinary state space exploration.

The advantage of the ComBack method is that it guarantees full coverage of the state space, unlike related methods such as hash compaction and bit-state

hashing. From a practical viewpoint one could therefore use methods such as hash compaction in early phases of a verification process to discover errors, and when no further errors can be detected, the ComBack method could be used for formal verification of properties.

In this paper we have not discussed verification of properties using the ComBack method. It can be observed that the method explores the full state space without mandating a particular exploration order. Furthermore, the state reconstruction that occurs when checking whether a state has already been visited can be made fully transparent to the verification algorithm being applied in conjunction with the state space exploration. This makes the method compatible with most on-the-fly verification algorithms (e.g., verification of safety properties and on-the-fly LTL model checking [166]). The ComBack method is also compatible with off-line verification algorithms such as CTL model checking [106] since the backedge table allows the reconstruction of any of the full state descriptors which in turn allows the forward edges between states to be reconstructed. Alternatively, we can simple store the forward edges in an additional table during state space exploration.

The ComBack method opens up several areas for future work. One topic is the integration of verification algorithms as sketched in the previous paragraph. Future work also includes implementation of the additional variants presented in Sect. 6.5, and the development and evaluation of caching strategies and organisation of collision lists to reduce the time spent on state reconstruction. It would also be interesting to compare the ComBack method to other complete techniques such as state caching [60]. Another important topic is to explore the combination of the ComBack method with other reduction methods. For this purpose, partial-order methods [134, 160] appear particularly promising as they reduce the in-degree of states which in turn will lead to a reduction in the number of state reconstructions.

Chapter 7

The BRITNeY Suite Animation Tool

The paper *The ComBack Method – Extending Hash Compaction with Backtracking* presented in this chapter has been published as a conference paper [T3]. The conference paper is a shortened and less technical version of the workshop paper [C6]. The BRITNeY Suite, described in these papers, is available from the BRITNeY Suite homepage [C2].

[T3] M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.

[C2] M. Westergaard. BRITNeY suite website. Online `wiki.daimi.au.dk/britney/`.

[C6] M. Westergaard and K.B. Lassen. Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY animation and CPN Tools. In *Proc. of Sixth CPN Workshop*, volume PB-576 of *DAIMI*, pages 119–136, 2005.

The version presented here is identical to the conference paper except for minor typographical changes.

The BRITNeY Suite Animation Tool

Michael Westergaard and Kristian Bisgaard Lassen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,k.b.lassen}@daimi.au.dk

Abstract

This paper describes the BRITNeY suite, a tool which enables users to create visualizations of formal models. BRITNeY suite is integrated with CPN Tools, and we give an example of how to extend a simple stop-and-wait protocol with a visualization in the form of message sequence charts. We also show examples of animations created during industrial projects to give an impression of what is possible with the BRITNeY suite.

7.1 Introduction

Colored Petri nets (CP-nets or CPN) [91] have proved their usefulness in modeling and understanding complex systems [10, 103, 112, 142], e.g., for verification of existing behavior or requirements engineering of needed behavior.

However, when using CP-nets, only people familiar with the formalism are able to truly understand the model of the system. A domain expert may understand a CP-net, when introduced to CP-nets in general and when the particular CP-net is explained by the model developer, but the domain expert is seldom able to talk back, say precisely what is wrong with the model, and offer suggestions to fix the model of the system, because of lack of technical expertise with the formalism. CP-net models of systems are prone to errors if they can not be fully understood and validated by someone with domain knowledge. The contribution of the BRITNeY[1] suite animation tool is to give a visualization of the state and actions of a CP-net so the domain expert can validate the model.

In this paper we present the BRITNeY suite [C2] which introduces an animation layer for CP-nets. BRITNeY suite provides a uniform way to implement, integrate, and deploy visualizations of CP-nets and has a pluggable architecture which makes it possible to write customized plug-ins to animate the model in addition to more than a dozen predefined plug-ins. The BRITNeY suite has already been used successfully to animate a network protocol [T4], to animate a workflow process in a bank for the purpose of requirements engineering [94] and to visualize how patient, nurse and doctor work together with a system that dispenses sedatives, again for the purpose of requirements engineering [114].

Even though BRITNeY suite is designed with CPN Tools in mind, it is possible to integrate the tool with any executable formalism as the interface to CPN Tools is based on well-known public standards. For example the tool has been used successfully to visualize the execution of a timed automaton [7] model as well as the reachability graphs of systems created using a subset of the π-calculus [127], bigraphical reactive systems [125], finite and timed automata,

[1] An abbreviation for **B**asic **R**eal-time **I**nteractive **T**ool for **Ne**t-based animation.

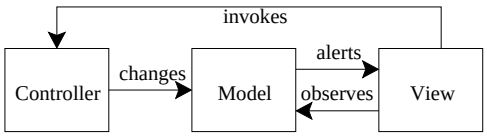


Figure 7.1: Architectural overview of the model-view-controller design pattern.

and Coloured Petri nets. Also goto-graphs of Java programs have been visualized using the BRITNeY suite.

The paper is structured as follows. In Sect. 7.2 we give a brief overview of the architecture of the BRITNeY suite. In Sect. 7.3 we demonstrate how to add a message sequence chart visualization to a CP-net of a simple stop-and-wait protocol. Sect. 7.4 contains some example visualizations created as part of industrial projects. In Sect. 7.5 we mention related work and outline some of the new features planned for BRITNeY suite.

7.2 Architectural Overview

A well-known design pattern from the object-oriented world is the model-view-controller (MVC) design pattern [54]. In the MVC design pattern, three participants collaborate to provide the implementation of an application, namely a model, a view, and a controller, see Fig. 7.1. The model contains the state of the system, the view is a (graphical) representation of the current state of the model, and the controller implements the behavior of the system. The view may initiate actions in the controller.

The idea behind the BRITNeY suite is to use a CP-net (or any other formal executable model) to model the state and behavior of the system (the model and controller), and use BRITNeY suite for visualizing the system (view). This division is natural as places of CP-nets are used to model the state of a system and transitions the behavior.

In Fig. 7.2, we see how BRITNeY suite is integrated with CPN Tools [33] to provide simulation-based visualizations and animations. CPN Tools itself is split into two components, an editor and a simulator. The animation tool, in the right part of the figure, communicates with CPN Tools using a standard Remote Procedure Call protocol, called XML-RPC [170], in order to allow vendors of other tools to directly integrate their tools with BRITNeY suite. BRITNeY suite uses plug-ins to make the actual visualizations, which makes it easy to create your own animations. 15 plug-ins are currently available in the tool. Table 7.1 lists each plug-in with a short description. Over time, more plug-ins will be added.

BRITNeY does not contain a fixed set of plug-ins as plug-ins can be added and removed, so stubs are generated on-the-fly as needed by using the reflection mechanism in Java to inspect the signatures of the plug-ins. The stubs make sure that values are passed correctly to the appropriate Java object's method and takes care of passing the return value back to the caller. Stubs are generated automatically by the stub generator component of BRITNeY suite. The stubs are injected into CPN Tools and are available as regular functions in the inscription language of CPN Tools, namely Standard ML (SML) [126], which allows the modeler to use the animation plugins anywhere SML expressions are allowed.

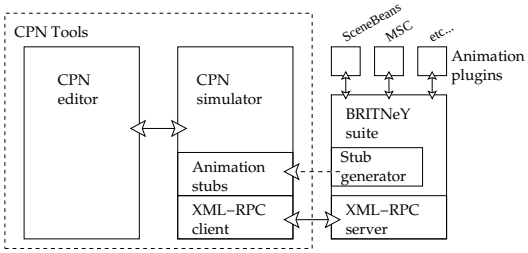


Figure 7.2: A more detailed view of the integration of the animation tool with CPN Tools.

Table 7.1: Plug-ins for the BRITNeY suite. The first group of plug-ins is for creating various charts, the second group is for displaying directed graphs, the third group is for interacting with a user and the final group contain plug-ins that do not fit in any group.

Name	Description
AreaChart	For visualizing data values by filling the area below them
GanttChart	For drawing Gantt charts
Histogram	For drawing histograms
MSC	For drawing message sequence charts
PieChart	For drawing pie charts
PieChart3D	For drawing 3D pie charts
StepChart	Similar to a histogram
XYChart	For visualizing data values as points
Graph	For drawing 2D graphs
Graph3D	For drawing 3D graphs
GetString	For getting short text-messages from the user
ShowString	Display short text-messages to the user
DataStore	Storage for simple data-types
Report	Nice presentation of data
SceneBeans	For displaying and interacting with a SceneBeans [149] animation

The modeler will often want to update the visualization when a transition occurs. This is done by calling the stubs in code segments that are special transition inscriptions allowed by CPN Tools. A code segment is executed when the transitions it belongs to occurs. It consists of input, output, and action parts. The input and output parts make it possible to receive input from the model and to provide situmli back to the model respectively. This makes it possible to, e.g., invoke a stub with values dictated by tokens and to generate new tokens from the result of executing the stub.

7.3 Using BRITNeY to Generate Message Sequence Charts

In this section we will describe how to show a simulation of a CP-net as a message sequence chart (MSC), i.e. generate a chart which displays the simulation of the CP-net in terms of events being passed between processes. This is

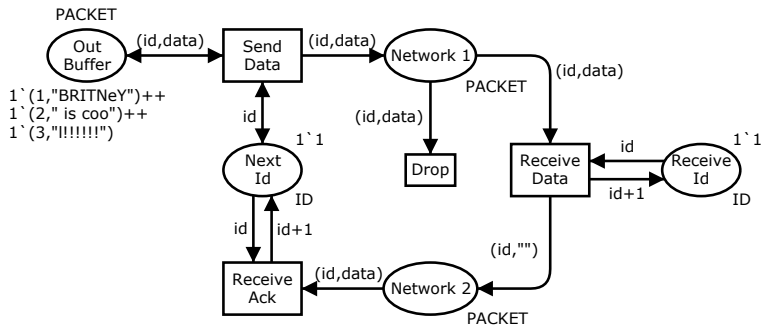


Figure 7.3: CP-net of a stop-and-wait protocol.

instead of, e.g., in CPN Tools where simulation is shown as enabling of transitions, and tokens being consumed and generated when transitions occur. The description is fairly high-level, and a more detailed and technical description can be found in [C6], but the reader is assumed to have basic knowledge of object oriented programming and an ability to read Java and SML code.

7.3.1 Model

The model that we will use in this paper is a very simple stop-and-wait protocol as seen in Fig. 7.3. The model consists of three parts: 1) A sender who can Send Data from the Out Buffer with a packet number from Next Id. Also the sender can Receive Ack thereby updating the token on Next Id. 2) A network that can Drop packets that are sent to the receiver from place Network 1. Network 2 contains acknowledgments that the receiver is sending back to the sender. 3) The receiver can Receive Data and update the Receive Id that the next packet must have.

7.3.2 Adding the MSC primitives in CPN Tools

MSCs are well-known to protocol engineers, and it is therefore a good idea to be able to present the execution of a CP-net as an MSC. The first part of an MSC that is generated from the model in Fig. 7.3 can be seen in Fig. 7.4. The Sender process corresponds to the sender part of the CP-net, Network process to the network part of the CP-net and Receiver process to the receiver part of the CP-net. In the following we will describe how to extend the model in Fig. 7.3 with primitives to draw this MSC.

In Listing 1 we show the signature of the Java plug-in for the MSC class. It contains functions for adding new process, adding events between processes, and adding events internal to a single process. This will, as explained in Sect. 7.2, be translated, by the stub generator, to a corresponding SML representation. In the following we show how to apply SML primitives to the CP-net to call these methods.

Listing 1 Java signature of the MSC object.

```
1 void addProcess(String name);
2 void addEvent(String from, String to, String name);
3 void addInternalEvent(String process, String name);
```

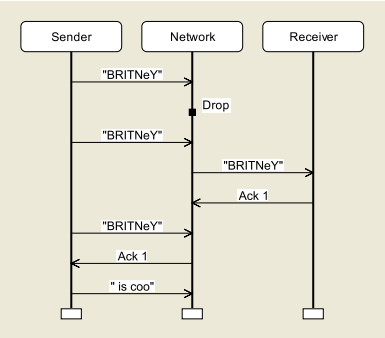


Figure 7.4: First part of an MSC generated from the model in Fig. 7.3

Listing 2 Initialization of the MSC view.

```
1 structure msc = MSC(val name = "Stop-and-Wait Protocol");
2 val _ = msc.addProcess("Sender");
3 val _ = msc.addProcess("Net");
4 val _ = msc.addProcess("Receiver");
```

To set up the MSC view we need to add some declarations to the CP-net. In CPN Tools we add declarations as in Listing 2. Line 1 initializes an MSC object with the name ”Stop-and-Wait Protocol”. Lines 2–4 creates the three processes as seen in Fig. 7.4; i.e. Sender, Network, and Receiver.

Next we need to extend our model from Fig. 7.3 to generate the events that correspond to those in Fig. 7.4.

In Fig. 7.5 we see how the methods from Listing 1 are incorporated into the CP-net. The idea is that we want to generate an event in the MSC when one of the transitions in the model occurs. We did this as follows: When Send Data in the CP-net occurs we add an event from Sender to Network in the MSC, where the label is the same as the data being sent, i.e. ”data” where data is bound from the string in the packet from Out Buffer. When Drop in the CP-net occurs we add the internal event Drop on the process Network in the MSC. When Receive Data in the CP-net occurs, an event is added from Network to Receiver in the MSC, with label stating what data is received (the label is ”data”, where data is bound from the string in the packet from Network 1) and also, an event from Receiver to Network in the MSC, with an acknowledgment with the received packet number as label; the label is Ack i where i is the integer in the packet bound in the occurence of Receive Data. Finally, when Receive Ack occurs, an event is sent from Network to Sender in the MSC with the acknowledgment as the label; here the label is again Ack i.

7.4 Visualization Examples

This section will give a number of examples of practical use of BRITNeY suite. We will not describe the examples in detail, but just refer to papers with detailed descriptions.

In Fig. 7.6, we see an animation created to visualize 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. The domain-specific GUI makes it possible for the

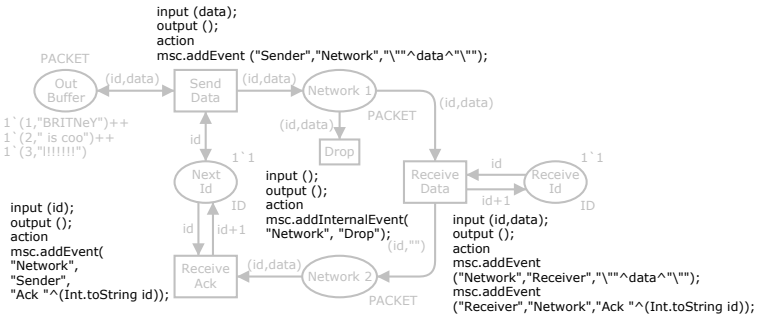


Figure 7.5: Model from Fig. 7.3 with MSC primitives

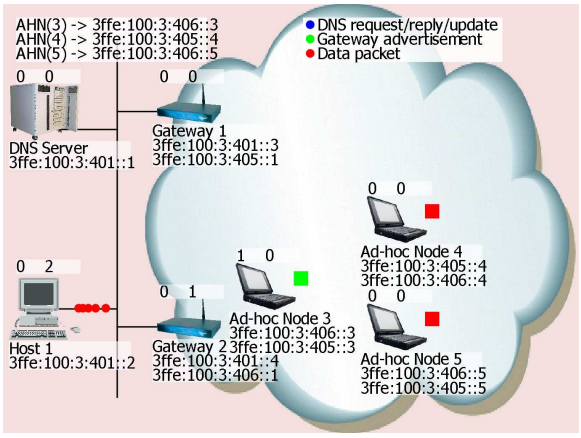


Figure 7.6: A visualization of an interoperability protocol for mobile ad-hoc networks.

user to observe the behavior of the system as packets, visualized by colored dots, flow along the network and to provide stimuli to the protocol by dragging and dropping the laptops to indicate the node movements. 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 modeling.

In Fig. 7.7 we see the domain specific animation based on the SceneBeans plug-in. This was used in [94] for requirements engineering of a new workflow system. The goal of the workflow was to support the handling of a blanc loan applications.

The animation is constructred as follows: There are always two bank assistants, Ann and Bill. Up to two customers can be present, in the figure only Mr. Smith is present. A bank manager, Mr. Banks, is always present. The balls represent blanc loan requests and the position of it shows who is responsible for the request. Whenever a transfer of responsibility occurs in the CP-net the ball is moved from one person to another in the animation. One ball has a P on it. This means that it is suspended, or parked, but can be picked up by one of the bank employees when they have the time. The square is part of the animation interface.

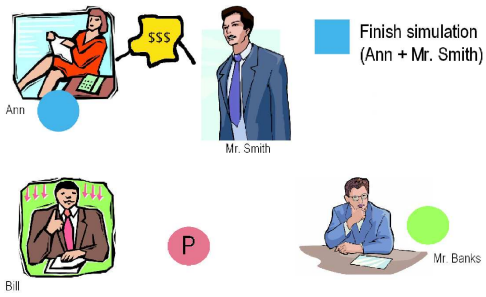


Figure 7.7: SceneBeans animation used for requirements engineering.

Once in a while the user can interact with the animation by e.g., setting up a loan for the customer when he wants to make a loan request, or setting the status of a loan request on behalf of Ann, Bill or Mr. Banks to e.g. granted or rejected. By not making the animation look like a normal prototype with windows, menues etc., the focus of the user was on the workflow and not on how the interface of the future system should be like.

7.5 Related Work and Future Improvements

BRITNeY suite supports adding animations to CPN models by annotating transitions with function calls, which are executed whenever the transition occurs. In the following, we outline how a number of other modeling tools facilitate visualization.

ExSpect [50], a tool for modeling based on CP-nets, allows the user to view the state by associating widgets with the state of the model, and to asynchronously interact with the model, also using simple widgets. In this way, it is easy to create simple user interfaces that support displaying information, but support for creating more elaborate animations is not readily available.

MIMIC/CPN [141] makes it possible to animate models in DESIGN/CPN [37], which is another tool for modeling using CP-nets. CPN models are animated by MIMIC/CPN by using function calls that are executed whenever a transition of the CP-net occurs. The animations are drawn using an application that resembles traditional drawing programs. Input from the user is possible by showing a modal dialog, where the simulation of the model is stopped while the user is expected to input information. It is also possible to make click-able regions, and the model can then query if one of these has been clicked. Another approach, which is taken by the COMMS/CPN [53] library for DESIGN/CPN and CPN Tools, is to provide a TCP/IP abstraction, allowing the user to code the user interface in any language and use RPC to communicate with it.

LTSA [116], a tool for modeling using timed labeled transition systems, allows users to animate models using the SceneBeans library. In LTSA animations are tied to the models by associating each animation activity with a clock; resetting a clock corresponds to starting an animation sequence. The animation sequence or a user with his mouse can then send events which correspond to the progress of the timer.

PNVis [99] is an add-on for the Petri Net Kernel [169], a highly modular tool for editing Petri nets. PNVis associates tokens with 3D objects and certain places with locations in a 3D world. Moving tokens corresponds to moving the associated object in the 3D world. PNVis is suitable for modeling physical sys-

tems, but not so applicable for creating prototypes of software or requirements engineering.

Using some of these animation tools/libraries, animation is integrated with the modeling formalism, such as the use of timers in LTSA or the ability to view or change the marking of places in ExSpect. Some libraries are easy to extend, such as animations in LTSA, as the SceneBeans library allows users to easily extend it with new animation primitives. Also, animations created using COMMS/CPN can easily be extended, as the "animation" is just a custom (e.g. Java) application. Some libraries make it easy to design animations, such as ExSpect and MIMIC/CPN, which both provide a graphical user interface to design animations. The approach of the current version of BRITNeY suite resembles a combination of MIMIC/CPN and COMMS/CPN, as the animation is driven by function calls associated with transitions to an external application. The main feature offered by BRITNeY suite from a user point of view is thus compatibility with CPN Tools (rather than the discontinued DESIGN/CPN) and platform-independence. BRITNeY suite also makes it easy to extend the tool using simple Java classes. From a developer point of view, BRITNeY provides good foundations for allowing closer integration with the model by allowing parts of the animation to inspect and modify tokens on fusion places of the CPN model, much like how widgets are associated with places in ExSpect. This is an important part of future work.

An important new feature of BRITNeY suite is that it is possible to deploy animations in a way that allows even non-technical users to download and experiment with the animation. Another part of the future work is to make this process even easier by adding a wizard to take care of all the details.

BRITNeY suite has already proven itself useful in real projects, and has already been used in several industrial projects.

Chapter 8

Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks

The paper *Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks* presented in this chapter has been published as a conference paper [T4].

[T4] L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.

The version presented here is identical to the conference paper except for minor typographical changes.

Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks

L. M. Kristensen*† M. Westergaard* P. C. Nørgaard‡

*Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {kris,mw}@daimi.au.dk

‡Ericsson Danmark A/S, Telebit,
Skanderborgvej 222, DK-8260 Viby J, Denmark,
Email: Peder.Chr.Norgaard@ericsson.com

Abstract

We present an industrial project conducted at Ericsson Danmark A/S, Telebit where formal methods in the form of Coloured Petri Nets (CP-nets or CPNs) have been used for the specification of an interoperability protocol for routing packets between fixed core networks and mobile ad-hoc networks. The interoperability protocol ensures that a packet flow between a host in a core network and a mobile node in an ad-hoc network is always relayed via one of the closest gateways connecting the core network and the mobile ad-hoc network. This paper shows how integrated use of CP-nets and application-specific visualisation have been applied to build a model-based prototype of the interoperability protocol. The prototype consists of two parts: a CPN model that formally specifies the protocol mechanisms and a graphical user interface for experimenting with the protocol. The project demonstrates that the use of formal modelling combined with the use of application-specific visualisation can be an effective approach to rapidly construct an executable prototype of a communication protocol.

Keywords: Model-driven prototyping; animation; Coloured Petri Nets; mobile ad-hoc network.

8.1 Introduction

The specification and development of communication protocols is a complex task. One of the reasons is that protocols consist of a number of independent concurrent protocol entities that may proceed in many different ways depending on when, e.g., packets are lost, timers expire, and processes are scheduled. The complex behaviour makes the design of protocols a challenging task. Protocols operating in networks with mobile nodes and wireless communication present an additional set of challenges in protocol engineering since the orchestration of realistic scenarios with many mobile nodes is impractical, and the physical characteristics of wireless communication makes reproduction of errors and scenarios almost impossible.

†Supported by the Danish Natural Science Research Council.

We present a case study from a joint research project [101] between the Coloured Petri Nets Group [34] at University of Aarhus and Ericsson Danmark A/S, Telebit [47]. The research project applies formal methods in the form of Coloured Petri Nets (CP-nets or CPNs) [91, 102] and the supporting CPN Tools [33] in the development of Internet Protocol Version 6 (IPv6) [82] based protocols for ad-hoc networking [137]. An ad-hoc network is a collection of mobile nodes, such as laptops, personal digital assistants, and mobile phones, capable of establishing a communication infrastructure for their common use. Ad-hoc networking differs from conventional networks in that the nodes in the ad-hoc network operate in a fully self-configuring and distributed manner, without any preexisting communication infrastructure such as base stations and routers.

CP-nets is a graphical discrete-event modelling language applicable for concurrent and distributed systems. CP-nets are based on Petri nets [143] and the programming language Standard ML (SML) [159]. Petri nets provide the foundation of the graphical notation and the basic primitives for modelling concurrency, communication, and synchronisation. The SML programming language provides the primitives for the definition of data types, modelling data manipulation, and for creating compact and parameterisable models. CPN models are executable and describe the states of a system and the events (transitions) between the states. CP-nets includes a module concept that makes it possible to organise large models into a hierarchically related set of modules. The CPN modelling language is supported by CPN Tools and have previously been applied in a number of projects for modelling and validation of protocols (see, e.g., [61, 64, 103, 132]).

The use of formal modelling languages such as CP-nets for specification and validation of protocols is attractive for several reasons. One advantage of formal models is that they are based on the construction of executable models that make it possible to observe and experiment with the behaviour of the protocol prior to implementation using, e.g., simulation. This typically leads to more complete specifications since the model will not be fully operational until all parts of the protocol have been at least abstractly specified. A model also makes it possible to explore larger scenarios than is practically possible with a physical setup. Another advantage of formal modelling is the support for abstraction, making it possible to specify protocols while ignoring many implementation details.

From a practical protocol engineering viewpoint, the use of formal modelling also have some shortcomings. Even if the modelling language supports abstraction and a module concept there is in most cases an overwhelming amount of detail in the constructed model. This is a disadvantage, in particular when presenting and discussing the design with colleagues unfamiliar with the applied modelling language. This means that a formal specification in many cases is accompanied by informal drawings being developed in parallel. The level of detail can also be a disadvantage when exploring the protocol design via, e.g., simulation. Furthermore, even if a model is executable, it still lacks the application- and domain-specific appeal of a conventional prototype.

The contribution of this paper is to present a model-based prototyping approach where formal modelling is integrated with the use of an animation GUI for visualising system behaviour to address the shortcomings of formal modelling discussed above. The approach has been applied to an interoperability protocol for routing packets between nodes in a mobile ad-hoc network and hosts in a fixed core network. Formal modelling is used for the specification of the protocol mechanisms and an application- and domain-specific GUI [C2] is added on top of the CPN model. The result is a *model-based prototype* in which

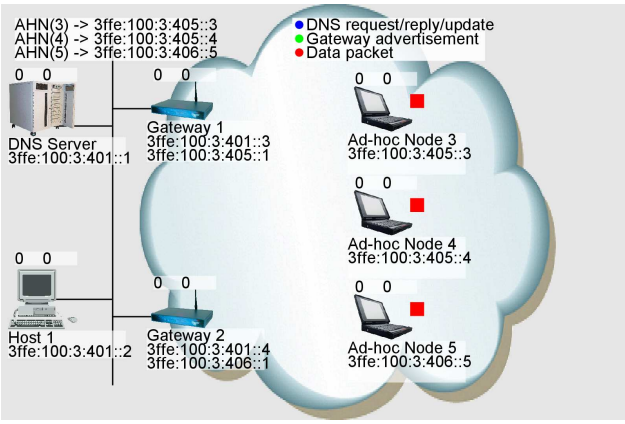


Figure 8.1: The hybrid network architecture.

the animation GUI makes it possible to observe the behaviour of the system and provide stimuli to the protocol. The use of an underlying formal model is fully transparent when experimenting with the prototype. The animation 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 rest of the paper is organised as follows. Section 8.2 gives a brief introduction to the network architecture and the interoperability protocol, and Sect. 8.3 presents the model-based prototyping approach. Section 8.4 presents selected parts of the CPN model specifying the interoperability protocol. Section 8.5 presents the graphical animation user interface and package applied in the project. Finally, Sect. 8.6 sums up the conclusions and presents related work.

8.2 The Interoperability Protocol

Figure 8.1 shows the hybrid network architecture captured by the model-based prototype. The network architecture consists of two parts: an IPv6 core network (left) and a mobile ad-hoc network (right). The core network consists of a Domain Name System (DNS) Server and Host 1. The mobile ad-hoc network contains three mobile nodes (Ad-hoc Node 3-5). The core network and the mobile ad-hoc network are connected by Gateway 1 and Gateway 2. A routing protocol for conventional IP networks (such as OSPF [110]) is deployed in the core network and a routing protocol for ad-hoc networks (such as OLSR [29]) is used in the mobile ad-hoc network. The purpose of the interoperability protocol is to ensure that packets are routed between hosts in the core network and nodes in the mobile ad-hoc network via the closest gateway.

The gateways periodically announce their presence to nodes in the mobile ad-hoc network by sending *gateway advertisements* containing an IPv6 *address prefix*. The address prefixes announced by the gateways are assumed to be unique, and the advertisement can be distributed to the ad-hoc nodes using, e.g., flooding. The interoperability protocol does not rely on a specific dissemination mechanism for the gateway advertisements. The interoperability protocol generalises to an arbitrary number of gateways and mobile nodes. Figure 8.1 shows the concrete setup represented in the model-based prototype.

IPv6 addresses [69] are 128-bit and by convention written in hexadecimal notation in groups of 16 bits separated by colon (:). Leading zeros are skipped within each group and a double colon (::) is a shorthand for a sequence of zeros. Addresses consists of an *address prefix* and an *interface identifier*. Address prefixes are written on the form x/y where x is an IPv6 address and y is the length of the prefix. The mobile nodes in the ad-hoc network configure IPv6 addresses based on the received gateway advertisements. In the network architecture depicted in Figure 8.1, Gateway 1 is announcing the 64-bit address prefix 3ffe:100:3:405::/64 and Gateway 2 is announcing the prefix 3ffe:100:4:406::/64. It can be seen from the labels below the mobile nodes that Ad-hoc Node 3 and Ad-hoc Node 4 have configured IP addresses based on the prefix announced by Gateway 1, whereas Ad-Hoc Node 5 has configured an IP address based on the prefix announced by Gateway 2. For an example, Ad-hoc Node 3 has configured the address 3ffe:100:3:405::3.

Each of the gateways has configured an address on the interface to the ad-hoc network based on the prefix they are announcing to the ad-hoc network. Gateway 1 has configured the address 3ffe:100:3:405::1 and Gateway 2 has configured the address 3ffe:100:3:406::1. The gateways have also configured addresses on the interface to the core network based on the 3ffe:100:3:401::/64 prefix of the core network. Host 1 in the core network has configured the address 3ffe:100:3:401::2 and the DNS server has configured the address 3ffe:100:3:401::1. The ad-hoc nodes may receive advertisements from both gateways and configure an IPv6 address based on each of the prefixes. The reachability of the address prefixes announced by the gateways in the ad-hoc network are announced in the core network via the routing protocol executed in the core network.

The basic idea in the interoperability protocol is that the mobile nodes register the IPv6 address in the DNS database which corresponds to the preferred (closest) gateway. Updates to the DNS database relies on the Dynamic Domain Name System Protocol [168]. The entries in the DNS database related to the mobile nodes are shown to the upper left in Figure 8.1. For an example, it can be seen that the entry for Ad-hoc Node 3 (AHN(3)) is mapped to the address 3ffe:100:3:405::3. When a mobile ad-hoc node discovers that another gateway is closer, it will send an update to the DNS server causing its DNS entry to be changed to the IPv6 address based on the prefix announced by the new gateway. It is assumed that the routing protocol executed in the mobile ad-hoc network will provide the information required for a mobile node to determine its distances to the currently reachable gateways. This means that when Host 1 wants to communicate, with e.g., Ad-hoc Node 3 and makes a DNS request to resolve the IP address of Ad-hoc Node 3, the DNS server will return the IP address corresponding to the prefix announced by the gateway closest to Ad-hoc Node 3.

8.3 Model-based Prototyping Methodology

Figure 8.2 shows the approach taken to use CPN models to develop a prototype of the interoperability protocol. A CPN model (lower left of Figure 8.2) has been developed by modelling the natural language protocol specification [130] (lower right) of the interoperability protocol. The modelling activity transforms the natural language specification into a formal executable specification represented by the CPN model. The CPN model captures the network architecture depicted in Figure 8.1 and the protocol mechanisms of the interoperability protocol, e.g., the periodic transmission of advertisements, the dynamic updates

of the DNS database, and traffic flows between hosts in the core network and nodes in the ad-hoc network. The resulting model can already be viewed as an early prototype since it is possible to execute and experiment with the protocol at the level of the CPN model. Since CP-nets is a graphical modelling language, it is possible to observe the execution of the model directly on the CPN model.

The CPN model provides a very detailed view on the execution of the system and it can be an advantage to provide a high-level way of interacting and experimenting with the prototype. Furthermore, when presenting the protocol design to people not familiar with CP-nets, it can be an advantage to be able to demonstrate the prototype without directly relying on the CPN model but more application and domain specific means. To support this, an *animation GUI* (top left of Figure 8.2) has been added on top of the CPN model. This graphics visualises the execution of the prototype using the graphical representation of the network architecture previously shown in Figure 8.1. The graphics is updated by the underlying CPN model according to the execution of the protocol.

In addition to observe feedback on the execution of the system in the animation GUI, it is also possible to provide input to the system directly via the animation GUI. In the prototype, it is possible for the demonstrator (e.g., a protocol engineer) to move the nodes in the ad-hoc network and to define traffic flows from the host in the core network to the nodes in the mobile ad-hoc network. The animation GUI has been implemented using a general visualisation package and framework [C2] developed in the course of the project (see Sect. 8.4).

Altogether the approach makes it possible to explore and demonstrate the prototype of the interoperability protocol based on the CPN model that formally captures the design, but doing it in such a way that the use an underlying formal model is transparent for the observer and the demonstrator.

8.4 The CPN Model

This section presents the CPN model specifying the interoperability protocol. The complete CPN model is hierarchically structured into 18 modules. As the CPN model is too large to be presented in full in this paper, we present only selected parts of the CPN model. The aim is to show how the key aspects of the interoperability protocol have been modelled. The key concepts of CP-nets

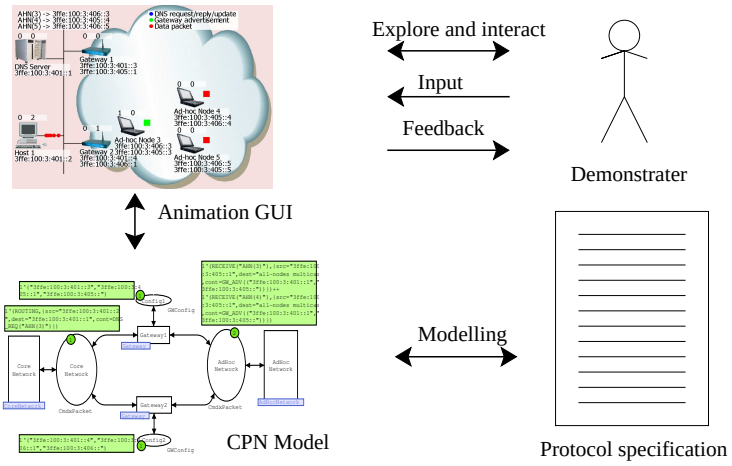


Figure 8.2: Model-based prototyping approach.

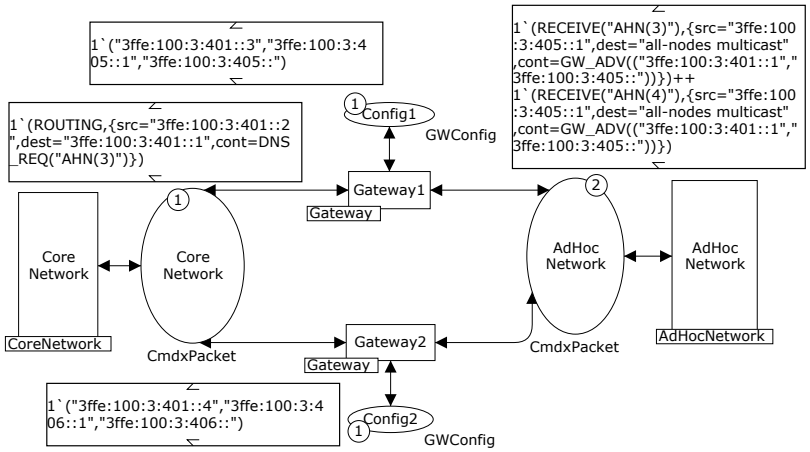


Figure 8.3: System module – top-level module of the CPN model.

will be briefly introduced as we proceed with the presentation. The reader is referred to [102] for a comprehensive introduction to CP-nets.

8.4.1 Model Overview

The module concept of CP-nets is based on the notion of *substitution transitions* which have associated *submodules* describing the compound behaviour represented by the substitution transition. A submodule of a substitution transition may again contain substitution transitions with associated submodules. Figure 8.3 shows the top level module of the CPN model which is composed of three main parts represented by the rectangular substitution transitions CoreNetwork (left), Gateway1 and Gateway2 (middle), and AdHocNetwork (right). The substitution transition CoreNetwork and its submodules model the core network, the substitution transition AdHocNetwork and its submodules model the mobile ad-hoc network, and the submodules of the two Gateway substitution transitions model the operation of the gateways connecting the core network and the mobile ad-hoc network. The text in the small rectangular box attached to each substitution transition gives the name of the associated submodule.

The state of a CPN model is represented through *places* (drawn as ellipses). There are four places in Fig. 8.3. The places CoreNetwork and AdHocNetwork are used for model modelling the packets in transit on the core network and ad-hoc network, respectively. The state of a CPN model is a distribution of tokens on the places of the CPN model. Figure 8.3 depicts a state where there is one token on place CoreNetwork and two tokens on place AdHocNetwork. The number of tokens on a place is written in the small circle attached to the place. The *data values* (colours) of the tokens are given in the filled box positioned next to the small circle. As an example, place CoreNetwork contains one token with the colours:

```
(ROUTING, {src="3ffe:100:3:401::2", dest="3ffe:100:3:401::1",
           cont=DNSREQ("AHN(3)")})
```

representing a DNS request in transit from Host 1 to the DNS server. Place AdHocNetwork contains two tokens representing gateway advertisements in transit to nodes in the ad-hoc network. The two Config places each contains a token representing the configuration of the corresponding gateway. It consists

of the IP address of the interface connected to the core network, the IP address of the interface connected to the ad-hoc network, and the prefix announced.

The data values (colours) of tokens that can reside on a place are determined by the *colour set* of the place which by convention is written below the place. Colour sets are similar to types known from conventional programming languages. Figure 8.4 lists the definitions of the colour sets (types) used in the System module. IP addresses, prefixes, and symbolic IP addresses are represented by colour sets IPAdr, Prefix, and Symname all defined as the set of strings. The colour set PacketCont and Packet are used for modelling the IP packets. The five different kinds of packets used in the interoperability protocol are modelled by PacketCont:

DNS_REQ modelling a DNS request packet. It contains the symbolic IP address to be resolved.

DNS_REP modelling a DNS reply. It contains the symbolic IP address and the resolved IP address.

DNS_UPD modelling a DNS update. It contains the symbolic IP address to be updated and the new IP address to be bound to the symbolic address.

GW_ADV modelling the advertisements disseminated from the gateways. An advertisement contains the IP address of the gateway and the announced prefix.

PACKET modelling generic payload packets transmitted between hosts and the mobile nodes.

The colour set Packet models the packets as a record containing the source, destination, and the content. The actual payload (content) and layout of packets are indifferent for modelling the interoperability protocol and has therefore been abstracted away. The colour set Cmd is used to control the operation of the various modules in the CPN model. The colour set GWConfig models the configuration information of the gateway.

8.4.2 Modelling the Core Network

Figure 8.5 shows the CoreNetwork module modelling the core network. This module is the immediate submodule of the substitution transition CoreNetwork of the System module shown in Figure 8.3. The *port place* CoreNetwork is assigned to the CoreNetwork *socket place* in the System module (see Figure 8.3). Port places are indicated by the In, Out, or I/O tags associated with them. The assignment of a port place to a socket place means that the two places are linked together and will always have identical tokens. By adding and removing tokens from port places, it is possible for a submodule to exchange tokens with its environment. The substitution transition Routing represents the routing mechanism in the core network. The substitution transition Host represents the host on the core network, and the substitution transition DNS Server represents the DNS server.

Hosts.

Figure 8.6 depicts the Host module modelling the host on the core network. The port place CoreNetwork (bottom) is assigned to the CoreNetwork socket place in the CoreNetwork module (see Figure 8.5). The module models the transmission of packets from the host to one of the mobile ad-hoc nodes. The substitution

```
(* --- Addressing --- *)
colset Prefix = string;  (* address prefixes *)
colset IPAdr = string;   (* IP addresses     *)
colset SymName = string; (* symbolic names   *)

colset SymNamexIPAdr = product SymName * IPAdr;
colset IPAdrxPrefix = product IPAdr * Prefix;

(* --- packets --- *)
colset PacketCont = union DNS_REQ : SymName +       (* DNS Request   *)
                          DNS_REP : SymNamexIPAdr + (* DNS Reply     *)
                          DNS_UPD : SymNamexIPAdr + (* DNS Update    *)
                          GW_ADV  : IPAdrxPrefix +  (* Advertisments *)
                          PACKET;                   (* Payload       *)

colset Packet = record src  : IPAdr  *
                       dest : IPAdr *
                       cont : PacketCont;

colset Cmd = union ROUTING +
                   RECEIVE      : IPAdr +
                   FLOODING     : IPAdr +
                   GWAHNROUTING : IPAdr +
                   AHNGWROUTING : IPAdr;

colset CmdxPacket = product Cmd * Packet;

(* --- Gateways configuration --- *)
colset GWConfig = product IPAdr * IPAdr * Prefix;
```

Figure 8.4: Colour set definitions used in the System module.

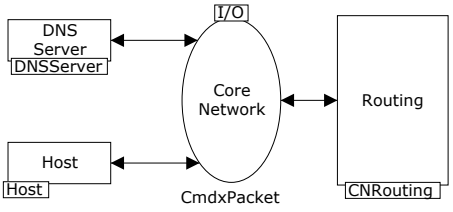


Figure 8.5: Core Network module – modelling the core network.

transition Flows (top) is used for interfacing with the animation GUI. We will return to this issue in Sect. 8.5.

The remaining places and transitions are used for modelling the behaviour of the host. The rectangles in Fig. 8.6 are ordinary transitions (i.e., not substitution transitions) which means that they can become *enabled* and *occur*. The dynamics of a CPN model consists of occurrences of enabled transitions that change the distribution of tokens on the places. An occurrence of a transition removes tokens from places connected to incoming arcs of the transition and adds tokens to places connected to outgoing arcs of the transition. The colours of the tokens removed from input places and added to output places are determined by *evaluating* the *arc expressions* on the arcs surrounding the transition. The arc expressions are written in the SML programming language. Data val-

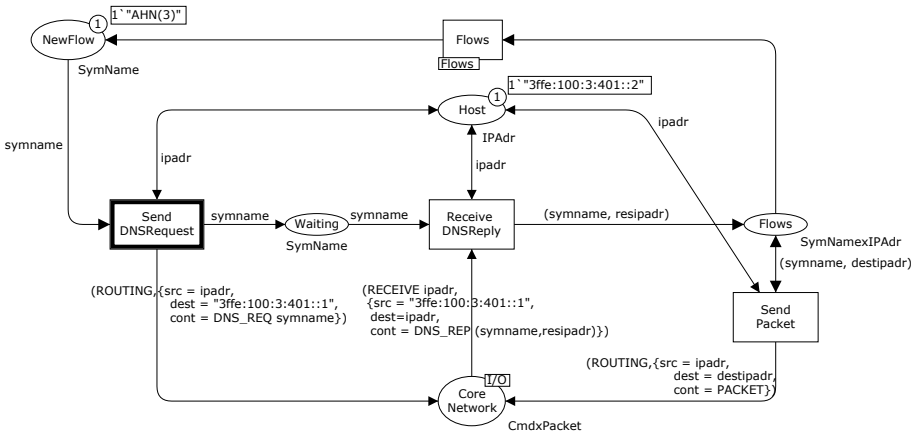


Figure 8.6: Host module – modelling the host.

ues must be bound to the *variables* appearing in the surrounding arc expressions before the arc expressions can be evaluated. This is done by creating a *binding element* which is a pair (t, b) consisting of a transition t and a *binding* b assigning data values to the variables of the transition. A binding element (t, b) is enabled iff the multi-set of tokens obtained by evaluating each input arc expression is a subset of the tokens present on the corresponding input place.

When the user defines a flow in the animation GUI, a token will appear in place NewFlow with a colour corresponding to the symbolic name of the mobile ad-hoc node which is the destination of the packet flow. An example is given in Fig. 8.6, where the NewFlow place contains a token corresponding to the user having defined a flow to Ad-hoc Node 3. This enables the SendDNSRequest transition in a binding where the value `"AHN(3)"` is bound to the variable `symname` of type SymName and the variable `ipadr` is bound to the value of the token on place Host specifying the IP address of the host.

When the SendDNSRequest transition occurs in the above binding, it will remove tokens from places `NewFlow` and `Host`, and add tokens to the output places Host, Waiting, and CoreNetwork. Tokens are added to the Host place since SendDNSRequest and Host are connected by a double arcs which is a short-hand for an arc in each direction having identical arc expressions. The colour of the tokens added are determined by evaluating the expressions on the output arcs. The resulting state is shown in Fig. 8.7. A token representing the IP address of the host is put back on place Host, a token representing the symbolic name to be resolved is put on place Waiting, and a token representing a DNS request has been put on place CoreNetwork.

The reception of the DNS reply from the DNS server is modelled by the transition ReceiveDNSReply which causes the token on place Waiting to be removed and a token to be added on place Flows. This corresponds to the host entering a state in which packets can be transmitted to the mobile ad-hoc node. The sending of packets is modelled by the transition SendPacket. The user may then decide (via the animation GUI) to terminate the packet flow which will cause the token on place Flows to be removed, and transmission of packets will cease. A host can have concurrent flows to different mobile ad-hoc nodes.

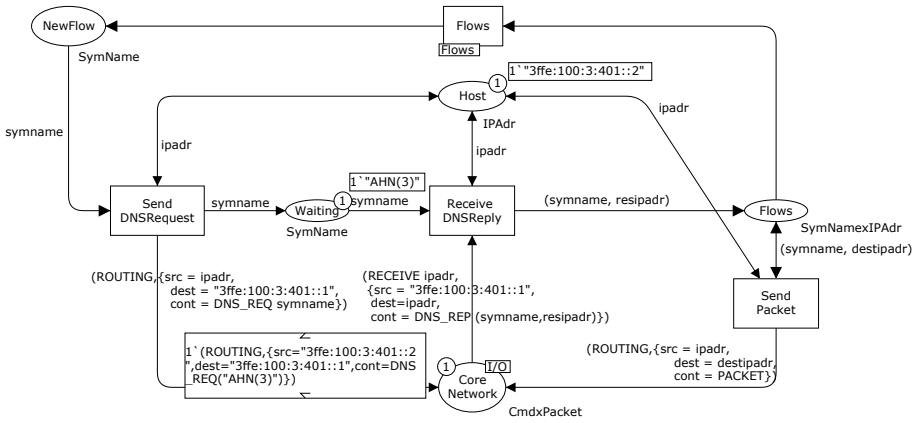


Figure 8.7: Host module – after occurrence of SendDNSRequest transition.

Domain Name Server and Database.

Figure 8.8 shows the DNSServer module modelling the DNS Server. The place DNSAdr contains a token corresponding to the IP address of the DNS Server. Place DNSDatabase models the DNS database entries on the DNS Server. There is a token on place DNSDatabase for each entry in the DNS database. The entries in the DNS database are modelled as tuples where the first component is the symbolic address (name) and the second component is the IP address bound to the symbolic name in the first component.

There are two possible events in the DNS server modelled by the transitions DNSRequest and DNSUpdate. The transition DNSRequest models the reception of DNS requests (from hosts) and the sending of the DNS reply containing the resolved IP address. The transition DNSUpdate models the reception of DNS updates from the mobile ad-hoc nodes. Both transitions access the DNSDatabase for lookup (transition DNSRequest) and modification (transition DNSUpdate).

Core Network Routing.

The CPN model does not specify a specific routing protocol but only the requirements to the core network routing protocol. This means that any routing protocol that meets these requirements can be used to implement the interop-

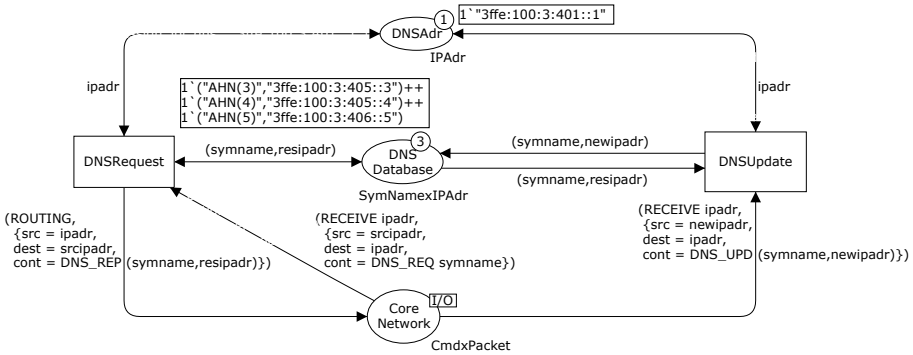


Figure 8.8: DNSServer module – modelling the DNS Server.

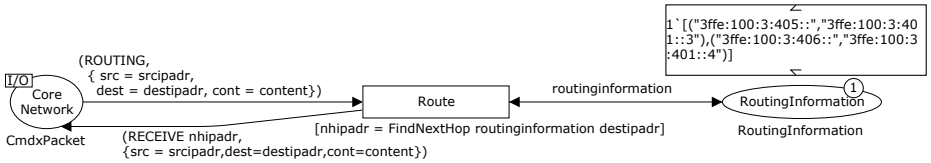


Figure 8.9: CNRouting module – Routing in the core network.

erability protocol. The routing mechanism in the core network is abstractly modelled by the CNRouting module shown in Figure 8.9. The place RoutingInformation models the routing information computed by the specific routing protocol in operation. This place contains a token that makes it possible given a prefix, to find the IP address of the corresponding gateway on the core network. This specifies the requirement that the gateways are required to participate in the routing protocol of the core network and announce a route to the prefix that they are advertising in the mobile ad-hoc network. This enables packets for nodes in the mobile ad-hoc network to be routed via the gateway advertising the prefix that matches the destination IP address of the packet. The transition Route models the routing of the packet on the core network. It uses the routing information on place RoutingInformation to direct the packet to the proper gateway. The function FindNextHop in the *guard expression* of the transition computes the IP address of the next hop gateway using the routing information and destination IP address of the packet.

8.4.3 Modelling the Gateways

The role of the gateway is to relay packets between the core network and the mobile ad-hoc network, and to periodically send advertisements to the mobile ad-hoc network. Figure 8.10 shows the Gateway module modelling the operation of the gateways. This module is the submodule of the two substitution transitions Gateway1 and Gateway2 on the System module. This means that there will be two *instances* of the Gateway module - one for each of the substitution transitions. Figure 8.10 shows the instance corresponding to Gateway1. The port place CoreNetwork is assigned to the socket place CoreNetwork and the port place AdHocNetwork is assigned to the socket place AdHocNetwork on the System module. The place Config contains a token giving the configuration of the gateway.

The relay of packets from the core network to the mobile ad-hoc network is modelled by the transition AHN_CoreTransmit and the relay of packets from the mobile ad-hoc network to the core network is modelled by the transition Core_AHNTransmit. Packets to be transmitted from the core network to the ad-hoc network are represented by tokens in the place CoreNetwork. When the transition Core_AHNTransmit occurs corresponding to the relay of a packet from the core network to the ad-hoc network, this token will be removed from the CoreNetwork place and a new token representing the packet added to the place AdHocNetwork. The relay of packets from the AdHocNetwork to the CoreNetwork is modelled in a similar manner by the transition AHN_CoreTransmit. The periodic transmission of advertisements on the mobile ad-hoc network is modelled by the substitution transition GatewayAdvertisement. The presentation of the submodule associated with this substitution transition has been omitted.

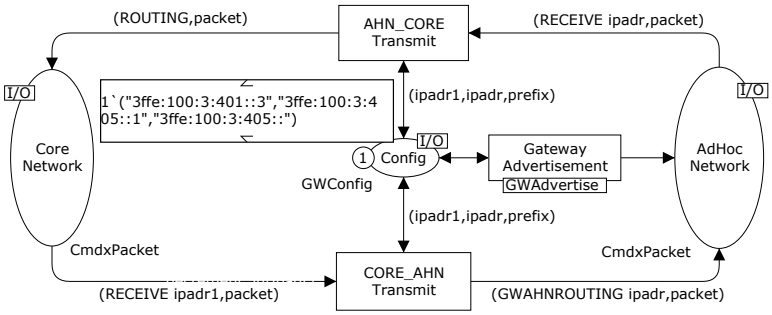


Figure 8.10: Gateway module – modelling the operation of the gateways.

8.4.4 Modelling the Mobile Ad-hoc Network

Figure 8.11 depicts the AdHocNetwork module which is the top level module of the part of the CPN model modelling the mobile ad-hoc network. The place Nodes is used to represent the nodes in the mobile ad-hoc network. The place RoutingInformation is used to represent the routing information in the ad-hoc network which is assumed to be available via the routing protocol executed in the ad-hoc network. This routing information enables among other things the nodes to determine the distance to the reachable gateways. Detailed information about the colour of the token on place RoutingInformation has been omitted.

Figure 8.12 lists the definition of the colour sets used in the AdHocNetwork module. The topology of the mobile ad-hoc network is abstractly represented by only representing the distance from each of the ad-hoc nodes to the two gateways. The reason is that it is only the relative distance to the two gateways which are of relevance to the operability protocol – not the complete topology. The colour set DistanceInformation is used to keep track of the reachability between the nodes in the ad-hoc network and the gateways. The distance information is a list with an entry for each pair of ad-hoc node and gateway. Each entry is again list consisting of a four-tuple (colour set DistanceEntry). Each entry consists of the symbolic name of the mobile ad-hoc node, its IP address (if configured), the IP address of the gateway (if configured), and the distance to the gateway. The gateway may also be unreachable in which case the distance is set to NOTREACH.

The colour set AHNConfig is used to model the configuration information

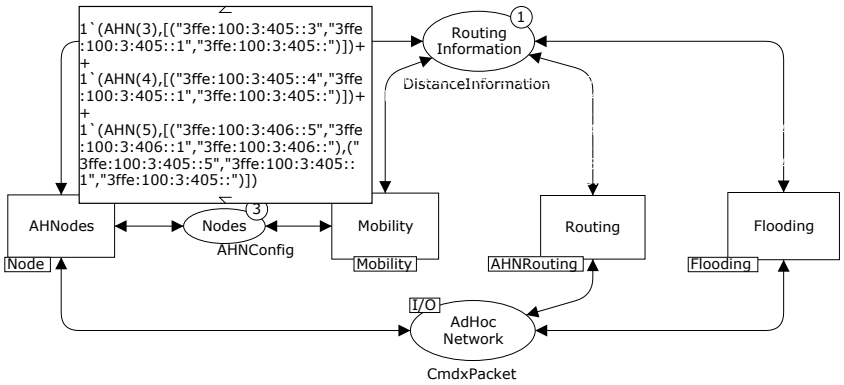


Figure 8.11: AdHocNetwork module – modelling the ad-hoc network.

```
(* --- ad-hoc nodes --- *)
colset AHId = int with 1..5;
colset AHNode = union AHN : AHId;

(* --- distance information --- *)
colset Distance = union REACH : Dist + NOTREACH;
colset DistanceEntry = product AHNode * IPAdr * IPAdr * Distance;
colset DistanceInformation = list DistanceEntry;

(* --- configuration information for ad-hoc nodes --- *)
colset AHNIPConfig = product IPAdr * IPAdr * Prefix;
colset AHNIPConfigs = list AHNIPConfig;
colset AHNConfig = product AHNode * AHNIPConfigs;
```

Figure 8.12: Colour definitions used in the AdHocNetwork module.

for the mobile ad-hoc nodes. Each ad-hoc node is represented by a token on place Nodes and the colour of the tokens specifies the name of the node and a list of configured IP addresses. Each configuration of an IP address specifies the IP address configured, and the IP address and prefix of the corresponding gateway. It is possible for a mobile ad-hoc node to configure an IP address for multiple gateways. The node will ensure that the DNS database always contains the IP address corresponding to the *preferred gateway*.

There are four substitution transitions in the AdHocNetwork module corresponding to the components of the ad-hoc network represented:

AHNodes represents the behaviour of the nodes in the mobile ad-hoc network. This will be presented in more detail below.

Mobility represents the mobility of nodes in the ad-hoc network, i.e., that the nodes may move closer or further away from the gateways. We will return to the modelling of mobility in Sect. 8.5.

Routing represents the routing protocol executed in the ad-hoc network. The purpose of the routing protocol in the context of the interoperability protocol is to provide the nodes with information about distances to the gateways. The routing is abstractly modelled in a similar way as the routing mechanism in the core network and will not be discussed further in this paper.

Flooding models the dissemination of advertisements from the gateways. A detailed presentation of this part of the model has been omitted.

Figure 8.13 depicts the Node module specifying the operation of the ad-hoc nodes. The module has three substitution transitions. PacketReceive represents the reception of packets from hosts in the core network. The submodule PacketReceive of this substitution transition is shown in Figure 8.14. The transition PacketReceive models the reception of a packet and consumes the token on place AdHocNetwork corresponding to the packet being received. AdvReceive represents the reception of advertisements from the gateways. A node changes its preferred IP address if the received advertisement is from a gateway which is closer than the gateway corresponding to the currently preferred gateway (if any). If the node configures a new preferred IP address based on the received advertisement, then it will send an update to the DNS server containing the new preferred IP address. DeleteGW represents the case where the

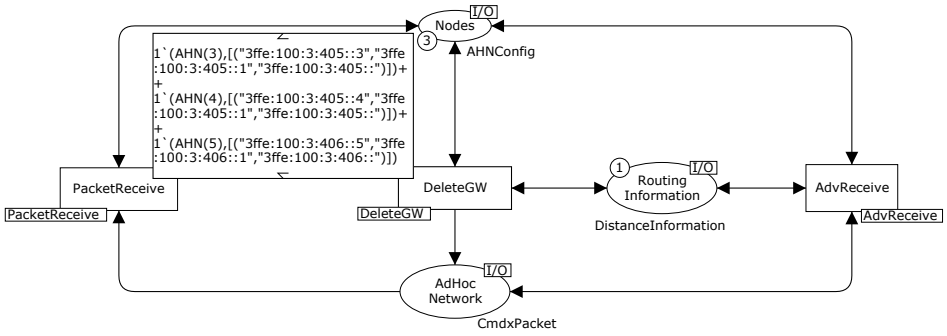


Figure 8.13: Node module – modelling an ad-hoc node.

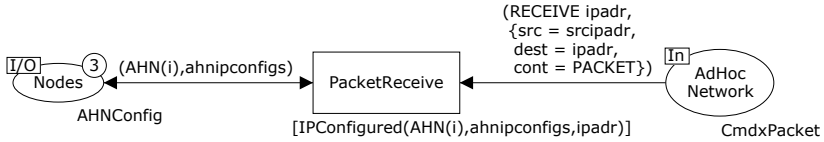


Figure 8.14: PacketReceive module – modelling reception of payload packets.

gateway corresponding to a configured IP address becomes unreachable. The assumption is that this will be detected via the routing protocol executed in the ad-hoc network or if advertisement has not been received for a specified amount of time. The submodules of the AdvReceive and DeleteGW are similar in complexity as the submodule of the PacketReceive substitution transition in Fig. 8.14 and has been omitted.

8.5 The Animation Graphical User Interface

The animation GUI has been implemented based on a general animation package [C2] developed in the course of the project. The animation package provides a general framework for adding various diagram types on top of executable models. The animation package is not designed specifically for CPN models, but is applicable also to other modelling formalisms.

The architecture of the model-based prototype developed in the project is depicted in Fig. 8.15 and consists of three main parts: The CPN Tools GUI (left), the CPN simulator (middle), and the animation GUI (right). The CPN Tools GUI and the CPN simulator constitute the CPN computer tools used in the project. CPN models are constructed using the CPN Tools GUI and the CPN simulator implements the formal semantics of CP-nets for execution of CPN models. The simulator communicates via the XML-RPC [170] infrastructure with the animation GUI to display the execution of the CPN model using the domain-specific graphics and for receiving stimuli/input from the demonstrator. The specific visualisation means are determined by the set of animation plug-ins used in the animation GUI. One animation plug-in was used to obtain *interaction graphics* in the form shown in Fig. 8.16. A second animation plug-in was used to obtain feedback in the form of message sequence charts (MSCs).

Figure 8.16 shows a representative snapshot of the application-specific during the execution of the CPN model. The IP addresses configured by the individual nodes are shown as labels below the nodes. For an example, Ad-hoc Node 3 has configured two IP addresses: 3ffe:100:3:405:3 and 3ffe:100:3:406:3.

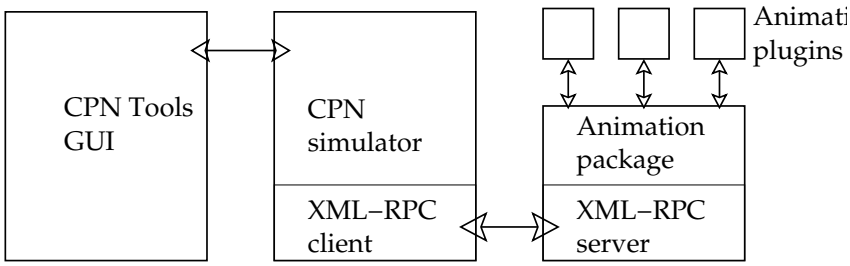


Figure 8.15: Architecture of the model-based prototype.

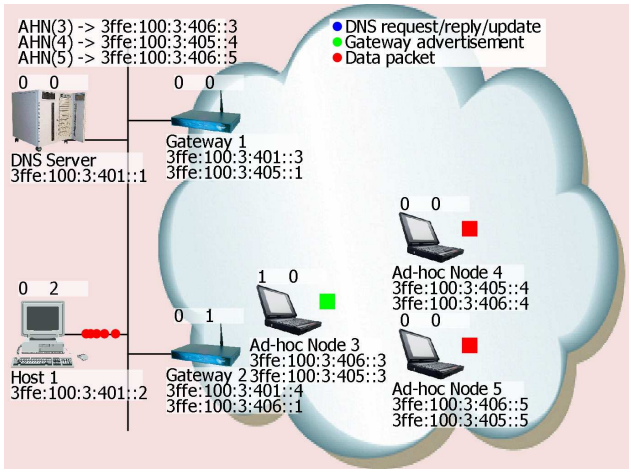


Figure 8.16: Snapshot of the interaction graphics.

The convention is that the preferred IP address is the topmost address in the list below the node. The entries in the DNS database are shown in the upper left corner. It shows the entries for each of the three ad-hoc nodes. The two numbers written at the top of each node are counters that provide information about the number of packets on the incoming (left) and outgoing (right) interfaces of the nodes. Transmissions of advertisements from the gateways are visualised by green dots. Transmission of payload packets are visualised using read dots, and DNS packets are visualised using blue dots. Figure 8.16 shows an example where Host 1 is transmitting a payload packet to Ad-hoc Node 3.

The user can move the nodes in the ad-hoc network thereby changing the distances to the two gateways. It is also possible to define a flow from the host in the core network to one of the nodes in the mobile ad-hoc network by clicking on the read square positioned next to each of the ad-hoc nodes. The square will change its colour to green once the CPN model has registered the flow. The flow can be stopped again by clicking on the (now green) square next to the mobile ad-hoc node. Finally, it is possible to force the transmission of an advertisement from a gateway by clicking on the gateway.

Figure 8.17 shows an example of a MSC creating based on a simulation of the CPN model. The MSC shows a scenario where Ad-hoc Node 3 makes a Move and discovers that Gateway 2 is now the closest gateway. This causes it to send a DNS update to the DNS server. The last part of the MSC shows the host initiating a packet flow to Ad-hoc Node 3.

Graphical feedback from the execution of the CPN model is achieved by

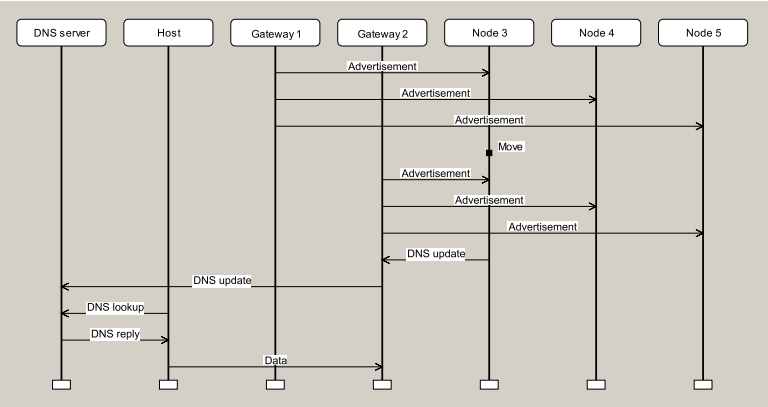


Figure 8.17: Message sequence chart generated by the animation GUI.

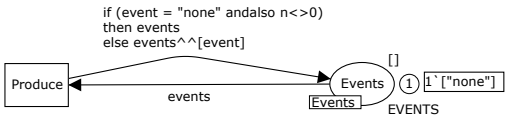


Figure 8.18: Poll module – Polling the animation GUI for events.

attaching *code segments* to the transitions in the CPN model. These code segments are executed whenever the corresponding transition occurs in the simulation/execution of the CPN model. As an example, the transition Route (see Figure 8.9) has an attached code segment which invokes the primitives required for animating the transmission of packets in the core network.

The CPN model receives input from the animation GUI by polling the animation GUI for events. The Poll module shown in Figure 8.18 polls the animation GUI for events at regular intervals during the execution of the CPN model and puts events into a list on the place Events, thereby implementing an event queue between the animation GUI and the CPN model. Parts of the CPN model that is to react on events from the animation GUI are linked to the Event place and are able to consume events from the event queue. The transition Produce polls the animation GUI for events.

8.6 Conclusions

We have presented our model-based prototype approach and demonstrated its use on an interoperability protocol. In addition to providing a detailed specification of the interoperability protocol via the constructed CPN model, the work has also highlighted the following characteristics and aspects of a model-based (virtual) prototyping approach:

Representation. The use of an animation GUI on top of the CPN model has the advantage 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. The disadvantage of this approach would have been a double representation of the dynamics of the interoperability protocol.

Transparency. The use of a domain specific graphical user interface (the animation GUI) has the advantage that the design can be experimented with and explored without having knowledge of the CPN modelling language. This has been shown in practise at a demonstration to management with no CPN knowledge.

Controllability. 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.

Abstraction. Implementation details can be abstracted away and only the key part 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, and the mechanism used for distribution of advertisements. Instead, we have modelled the service provided by these components. The possibility of making abstraction means that it is possible to obtain an executable prototype without implementing all components.

Feasibility. The use of a model means that there is no need to invest in physical equipment and there is no need to setup the 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.

Related Work

Integrated use of visualisation and formal modelling has also been considered for CP-nets in earlier work in the area of embedded systems [142], telecommunication protocols [14], pervasive electronic patient records [10], and software for mobile phones [112]. The case studies in [10, 14, 112, 142] all applied the MIMIC/CPN [141] package, an internal part of the DESIGN/CPN [37] tool. The approach presented in this paper relies on an external application handling the visualisation, which we find is a more flexible approach as it allows us to use existing software libraries supporting different diagram types. In MIMIC/CPN, input from the user is only possible by showing a modal dialog, meaning the simulation of the model is stopped while the user is expected to input information. The animation package presented in this paper avoids this by using an asynchronous event queue polled by a transition in the model. As part of future work, we plan to eliminate polling by allowing external applications to directly produce and consume tokens on special *external places*.

Visualisation is also available in other tool sets. ExSpect [50] allows the user to view the model state by associating widgets with the state of the model and asynchronously interact with the model using simple widgets. In this way, one creates simple user interfaces for displaying information and simple interaction. LTSA [116] allows users to animate models using an animation library called SceneBeans [117]. In LTSA animations are tied to the models by associating each animation activity with a clock; resetting a clock corresponds to starting an animation sequence, and events in the animation corresponds to progress of the clock. PNVis [99] associates objects of a 3D world with tokens, and is suitable for modelling physical systems, but not immediately applicable for network protocols. The Play-Engine [66] supports the developer in implementing a prototype by inputting scenarios (play-in) via an application-specific

GUI, and then execute the resulting program (play-out). Compared to our approach this makes the model implicit as the model is created indirectly via the input scenarios. We view an explicitly created model as an advantage when the prototype is to serve as a basis for an actual implementation of the system. The reason is that an implicitly created model is difficult to interpret as it is automatically generated.
In conclusion, the work presented in this paper has demonstrated that using CP-nets and the supporting computer tools for building a model-based prototype can be 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 dissemination 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.

Acknowledgements. The authors gratefully acknowledge the support of their colleagues in BAE SYSTEMS plc, Ericsson Microwave Systems AB and Ericsson Danmark A/S, Telebit, and support from the UK, Swedish and Danish MoDs under the EUCLID/Eurofinder programme, Project RTP6.22 (B2NCW). The authors would also like to acknowledge Rolf Christensen for his contributions.

Chapter 9

A Game-theoretic Approach to Behavioural Visualisation

The paper *A Game-theoretic Approach to Behavioural Visualisation* presented in his paper has been submitted to the 2^{nd} International Workshop on Formal Methods for Interactive Systems. The paper is a rewritten version of the workshop paper [C4]. The original workshop paper, [C4], focuses on the introduction of game coloured Petri nets, whereas the revised paper, [T5], focuses on a general formal framework for visualisations of formal models.

[T5] M. Westergaard. A Game-theoretic Approach to Behavioural Visualisation. Submitted, 2007.

[C4] M. Westergaard. Game Coloured Petri Nets. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 281–300, 2006.

The version presented here is identical to the submitted paper except for typographical changes.

A Game-theoretic Approach to Behavioural Visualisation

Michael Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: mw@daimi.au.dk

Abstract

To bridge the gap between domain experts and formal methods experts, visualisations of the behaviour of formal models are used to let the domain expert understand and experiment with the formal model. In this paper we provide a definition of visualisations, founded in game-theory, which regards visualisations as transition systems synchronised with formal models. We show example visualisations, use them to show winning strategies of games, and demonstrate how an industrial application of formal models benefited from this approach.

9.1 Introduction

Formal models are being used for specification and verification of complex systems [T4, 10, 61, 64, 94, 103], provide valuable insight into the workings of the systems, and may detect errors early in the development process. One problem of constructing formal models of systems is that the domain experts, who have a lot of knowledge of the domain of the modelled system, typically have little or no knowledge of formal models. At the same time, experts in formal models typically have little knowledge of the system domain. One way to solve this is to let the domain expert describe the system to the formal methods expert, who then constructs a model for specification and validation. The drawback of this approach is that it is very difficult to know whether problems in the model represent errors in the model itself or in the modelled system. The formal methods expert typically does not know the domain well enough to make the judgement for subtle errors, and the domain expert does not understand the formal model or the error report well enough to make the judgement either. One way to facilitate the communication between the formal methods expert and the domain expert is to create a domain-specific visualisation, which the domain expert can inspect and stimulate. Examples of visualisations include cartoon-like representations of, e.g., computers on a network and how they communicate or a live updated UML sequence diagram [131] showing how messages are exchanged between people working in a bank. We also provide an example of how this can be used to visualise problems found in a model.

In order to facilitate communication of formal models, several tools [T3, 50, 66, 99, 117, 141, 149] have been conceived with the purpose of constructing domain-specific visualisations. These tools rely on the methodology depicted in Fig. 9.1. Here a domain expert writes a specification of the system. The specification is usually written in natural language and only uses semi-formal

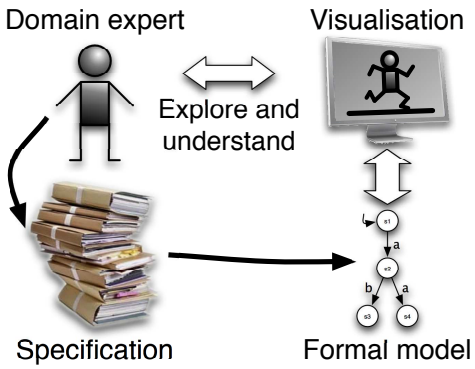


Figure 9.1: Methodology.

notation. Often the specification is vague and incomplete, maybe even self-contradictory. In order to make the specification clear, complete, and consistent, a formal executable model in constructed. This is usually done by a formal methods expert, who is not an expert on the domain. Ambiguities can be resolved during the construction of the model, which in itself makes the construction worthwhile. In order to ensure that the formal model actually reflects the specification, a visualisation is created. The behaviour of the visualisation is defined by the formal model, so the visualisation reflects the state of the model and changes in the model is reflected as updates to the visualisation. The domain expert is now able to see and understand what happens in the formal model, and can even interact with it. Inconsistencies between the model and the specification can be resolved as the domain expert identifies things that do not work as intended. We may then verify properties of the model required by the domain, e.g., that a network protocol cannot cause dead-locks, knowing that errors in the model probably reflect errors in the specification. One problem with tools facilitating the methodology in Fig. 9.1 is that they are built in an ad-hoc manner, as an afterthought, when the gap between domain experts and formal method experts becomes evident to researchers working with a specific formalism. Therefore the tools either mainly allows simple inspection of the state of the model during execution or require that the modeler spends a lot of time constructing a visualisation and integrating it with the model.

In this paper we propose a new, theoretically well-founded way to view visualisations, a declarative way for tying visualisations to a formal model, and a way to visualise error reports from formal verification so the domain expert is able to understand them. The idea is to view visualisations as transition systems, where the state of the system is what is visible to observers and labels on transitions are changes to what is visible. We tie visualisations to models by defining a synchronisation constraint [2], and require that the visualisation is able to simulate [124] the model, to make the behaviour of the synchronised product unconstrained by the visualisation and dictated by the model. This formulation only allows the domain expert to observe the behaviour of the formal model. In order to also allow the domain expert to provide input to the formal model, we regard it as describing a game between the modelled system and its surroundings; the domain expert then controls the environment and a computer tool controls the modelled system. The definition is formal and general, so it is possible to implement the method in computer tools supporting any formalism using transition systems as semantical foundation.

The paper is structured as follows: Sect. 9.2 describes related work, and in Sect. 9.3 some theoretical background material needed to understand the rest of the paper is provided. Sect. 9.4 introduces and exemplifies the idea of

regarding visualisations as transition systems synchronised with formal models, and in Sect. 9.5 two example uses of visualisations are described, namely a way to show winning strategies of games and an industrial application of the method in Fig. 9.1. In Sect. 9.6, we sum up our conclusions.

9.2 Related work

Several tools supporting the methodology in Fig. 9.1 exist. In this section we will describe some of them and discuss strengths and weaknesses of each.

ExSpect [50], a tool for modeling based on coloured Petri nets [91], allows the user to view the state of models by associating widgets with the state of the model, and allows users to asynchronously interact with the model using simple widgets. The disadvantage of this approach is, firstly, that it is specific to coloured Petri nets (as it relies on the special kind of state in a coloured Petri net) 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.

The BRITNeY Suite [C2, T3] and Mimic/CPN [141] are libraries which facilitate visualisation of coloured Petri net models. They provide an API which can be used to define and update visualisations. By annotating a model, these functions are called during execution of the model. 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, these tools mainly focus on the 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. Finally, these tools are unable to handle asynchronous input, which must be simulated by polling.

LTSA [116], a tool for modeling using timed transition systems, allows users to animate models using a library called SceneBeans [117, 149]. Animations are tied to models by associating animation activities with clocks. Resetting a clock corresponds to starting an animation sequence. The termination of an animation sequence, or a user with a mouse, sends events which correspond to progress of timers. The method is nice and declarative, but requires that we have clocks at our disposal, limiting the method to timed formalisms.

PNVis [99] is an add-on for the Petri Net Kernel [169], a modular tool for editing Petri nets [138]. PNVis associates 3D objects and locations in a 3D world with certain aspects of the state of the model and is hence suitable for modeling physical systems, but not aimed at systems that do not immediately have a physical counter-part.

The 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.

9.3 Theoretical background

Most of the work described in this paper has been developed in the context of coloured Petri nets (CP-nets or CPNs) [91] and game coloured Petri nets (game CPNs or game CP-nets) [C4], but applies to many other formalisms. In order to reflect that, we formulate our method using transition systems, which constitute the semantical foundation for several important modelling languages, e.g. CP-nets and the π-calculus [124]. In this section we recall definitions of transition systems, synchronised products, simulations, game transition systems (games), as well as winning strategies for games.

Definition 9.1 ((Labelled) Transition System) *A **transition system** (TS) is a tuple (S, T, δ, s_I), where S is a (finite or infinite) set of **states**, T is a (finite or infinite) set of **transitions**, $\delta \subseteq S \times T \times S$ is the **transition relation**, and $s_I \in S$ is the **initial state**.*

Four examples of transition systems can be seen in Fig. 9.2. Here we have represented each state by a circle and transitions as arcs leading from one circle to another. If there is an arc, labelled by a, leading from a circle labelled $s1$ to a circle labelled $s2$, it represents a transition $(s1, a, s2) \in \delta$. The initial state is marked by an incoming arc with no source. We will later explain why some arcs are dashed and some states are drawn using a double line. As an example, TS (a) can be represented as $\mathcal{TS} = (S, T, \delta, s_I)$ where $S = \{s1, s2, s3, s4\}$, $T = \{a, b\}$, $\delta = \{(s1, a, s2), (s2, b, s3), (s2, a, s4)\}$, and $s_I = s1$.

Let $\mathcal{TS} = (S, T, \delta, s_I)$ be a transition system, $s, s' \in S$ two states, and $t \in T$ a transition. If $(s, t, s') \in \delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. A *finite occurrence sequence*, σ, is an alternating sequence of states, s_i, and transitions, t_i, written $\sigma = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$ where $(s_i, t_i, s_{i+1}) \in \delta$ for $i = 1, \ldots, n-1$, and $s_1 = s_I$. An *infinite occurrence sequence*, σ', is an alternating sequence of states, s_i, and transitions, t_i, written $\sigma' = s_I = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} \cdots$ where $(s_i, t_i, s_{i+1}) \in \delta$ for $i \geq 1$, and $s_1 = s_I$. We denote by Σ^ω the set of all (finite and infinite) *occurrence sequences*.

We often wish to synchronise two or more transition systems, and a way to that is by forming a synchronised product by using a relation on transitions to define which must occur simultaneously, as formalised in Def. 9.2.

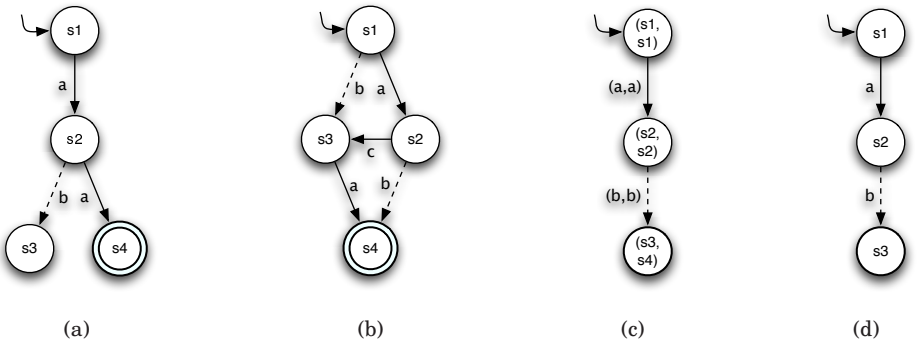


Figure 9.2: Four transition systems. (c) is a synchronisation of (a) and (b), (d) is equal to (c) except that its states and transitions have been renamed.

Definition 9.2 (Synchronised Product, [2]) *Let $\mathcal{TS}_i = (S_i, T_i, \delta_i, s_{Ii})$ for $i = 1, \ldots, n$ be transition systems. A **synchronisation constraint** is a relation $\mathcal{S} \subseteq T_1 \times T_2 \times \cdots \times T_n$. The **synchronised product** of $\mathcal{TS}_i$ w.r.t. $\mathcal{S}$ is $\mathcal{TS} = (S, T, \delta, s_I)$ with $S = S_1 \times S_2 \times \cdots \times S_n$, $T = \mathcal{S}$, $\delta = \{((s_1, \ldots, s_n), (t_1, \ldots, t_n), (s_1{}', \ldots, s_n{}')) \mid (t_1, \ldots, t_n) \in T, (s_i, t_i, s_i{}') \in \delta_i \text{ for } i = 1, \ldots, n\}$, and $s_I = (s_{I1}, s_{I2}, \ldots, s_{In})$.*

If we synchronise the TS (a) and TS (b) in Fig. 9.2 using the synchronisation constraint $\mathcal{S} = \{(a, a), (b, b)\}$, we obtain TS (c) (we have omitted states that are not reachable from the initial state). We notice that it is not possible for one of the TS to take a step autonomously using the above definition. We can simulate this by adding a distinguished transition Δ which leads from each state to itself. In the case of the TS in Fig. 9.2(a), we would add Δ to T and $\{(s1, \Delta, s1), (s2, \Delta, s2), (s3, \Delta, s3), (s4, \Delta, s4)\}$ to δ.

We often need to state that two transition systems behave in a similar way. We do this by defining a simulation, which states that one TS is able to exhibit the same behaviour as another (but not necessarily the other way around).

Definition 9.3 ((Strong) simulation [124]) *Let $\mathcal{TS}_i = (S_i, T, \delta_i, s_{Ii})$ for $i = 1, 2$ be transition systems sharing transitions. A relation $\preceq \subseteq S_1 \times S_2$ is a **simulation** iff whenever two states are in the relation, $s_1 \preceq s_2$, then for all transitions $\alpha \in T$, such that $s_1 \xrightarrow{\alpha} s_1{}'$, there exists a $s_2{}' \in S_2$ such that $s_1{}' \preceq s_2{}'$ and $s_2 \xrightarrow{\alpha} s_2{}'$. We say that $\mathcal{TS}_2$ **simulates** $\mathcal{TS}_1$ if there exists a simulation $\preceq \subseteq S_1 \times S_2$ such that $s_{I1} \preceq s_{I2}$.*

In Fig. 9.2 both (a) and (b) can simulate (d) using the simulations $\preceq_a = \{(s1, s1), (s2, s2), (s3, s3)\}$ respectively $\preceq_b = \{(s1, s1), (s2, s2), (s3, s4)\}$.

If we look at a game like tic-tac-toe, we see that it has two players, cross and naught. From the point of view of cross, it is only possible to add crosses to the board, naughts are added "automatically" according to the rules of the game. We want to reflect this in a transition system, so we split the transitions into two disjoints sets: the transitions controllable by the system we are modelling, and the transitions executed by the environment. We make the assumptions about the surroundings explicit in the model, yet provide a clear distinction between assumptions about the surroundings and the specification of the system. In the tic-tac-toe example, the action of adding a cross to the board is controllable by the modelled system and the action of adding a naught is not. Applying this to formal modelling, transitions of the modelled system are controllable, e.g., the actions of a network protocol, such as transmitting a packet or incrementing a counter, are controllable, whereas actions of the surroundings (e.g., a network), such as transmitting or altering a packet, are uncontrollable. Transitions of the environment formalise the assumptions about the surroundings (e.g. whether the network is allowed to alter packets). In normal games, like tic-tac-toe, we often also have some goal, e.g., ending up with three crosses in one row. This is also the case when modelling systems as games; in the case of a network protocol, a goal may be to successfully receive all packets in the correct order. A game is a TS where transitions are separated into disjoint sets: controllable and uncontrollable. Additionally we add a set of winning (goal) states. This is summarised in Def. 9.4.

Definition 9.4 (Game) *A **game** (or game transition system) is a tuple $(S, T^u, T^c, \delta, s_I, W)$, such that T^u is a set of **uncontrollable transitions** and T^c is a set of **controllable transitions** such that $T^u \cap T^c = \emptyset$, $W \subseteq S$ is a set of **winning states**, and $(S, T^u \cup T^c, \delta, s_I)$ is a transition system.*

We can turn any TS in Fig. 9.2 into a game by splitting the transitions into controllable and uncontrollable transitions and deciding which states are winning. For example, if we take $T^c = \{a\}$, $T^u = \{b\}$, and $W = \{s4\}$ we obtain a game for TS (a). In the figure we have shown uncontrollable transitions using dashed arcs. States in W are drawn using double lines.

A *strategy* is a function assigning to each state a controllable transition (if no controllable transitions are enabled in a given state, we can just map the state to any of the controllable transitions or add a distinguished transition Δ to T^c signifying "do nothing"). A winning strategy is a strategy, that ensures we always end up in a winning state, irregardless of what uncontrollable moves are chosen, i.e., a winning strategy is a "program" ensuring we and up in a good state. Formally:

Definition 9.5 (Winning Strategy) *Let* $(S, T^u, T^c, \delta, s_I, W)$ *be a game and* $\mathscr{S} : S \to T^c$ *a strategy. An occurrence sequence*

$\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n(\cdots) \in \Sigma^\omega, t_i \in T^u \cup T^c$ *is* ***consistent*** *with the strategy iff* $t_i \in T^c \implies t_i = \mathscr{S}(s_i)$ *for all* $i = 1, \ldots, n(, \ldots)$. *An occurrence sequence,* σ*, is* ***maximal*** *iff it is a) infinite, or b) finite,* $\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n$*, and if* $s_n \xrightarrow{t} s$ *for any* $s \in S$ *then* $t \in T^u$. *A strategy is a* ***winning strategy*** *iff all maximal occurrence sequences,* $\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n(\cdots) \in \Sigma^\omega$ *with* $s_1 = s_I$ *that are consistent with the strategy satisfy* $\exists k \geq 1$ *such that* $s_k \in W$.

If we take $T^c = \{a\}$, $T^u = \{b\}$, and $W = \{s4\}$ in Fig. 9.2(a), it is not possible to obtain a winning strategy (the only strategy is the mapping from all states to the transition a. The occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$ is consistent with that strategy, but does not lead to $s4$) whereas we can obtain a winning strategy in (b) using $T^c = \{a, c\}$, $T^u = \{b\}$, and $W = \{s4\}$ (the strategy mapping states $s1$, $s3$, and $s4$ to the transition a and $s2$ to c is winning).

In [18], an algorithm from [111] is instantiated to obtain an efficient (and optimal) algorithm to decide whether a given finite game (i.e. a game where $|S| + |T^u| + |T^c| < \infty$) has a winning strategy and to extract that strategy. The intuition of the algorithm is to calculate a minimal fix-point of all *good states*, where all states in W are good and all states where we can take a controllable step to a good state and all uncontrollable steps leading to a good state are good.

In Fig. 9.2(a), the only state which can be marked as good is $s4$ ($s3$ is not good as it has no successors, $s2$ not good as the b transition leads to $s3$, which is not good, and in $s1$ a leads to $s2$, which is not good). In (b), initially $s4$ is good. We can then mark $s3$ as good (as we can take an a transition to $s4$, which is good). After that, we can mark $s2$ as good (c, which is controllable, leads to $s3$ and b, all uncontrollable transitions enabled in $s2$, lead to $s4$). Finally we can mark $s1$ as good as both a and b lead to good states.

Using this algorithm, we can obtain a winning strategy for any game (if one exists). Often we are not satisfied knowing whether a winning strategy exists. If one does, we are interested in obtaining the winning strategy, as we can use as a guide to execute our model so we reach a winning state. We often require a counter example if no winning strategy exists so we can understand why. Until now, when concluding that a given game does not have a winning strategy, the best counter example we could provide was a list of all good states. This can be useful for small examples, but for systems with millions or more states this is not very useful. The purpose of the counter example is to convince a user that it is not possible to have a winning strategy. If a user needs conviction, it is probably because he thinks he knows a winning strategy. In this paper we will

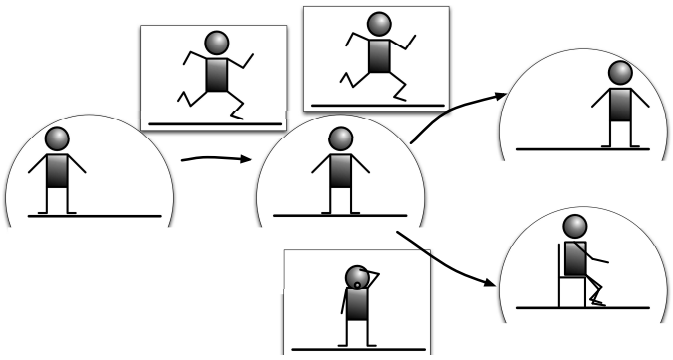


Figure 9.3: Visualisations as transitions systems.

propose a new way of providing counter examples to the existence of winning strategies. We let the user assume the role of the modelled system and let him try out his winning strategy against the computer, which knows how to counter all moves of the user. The user will try his winning strategy on a visualisation of the model. We will go into more detail about this in Sect. 9.5.2.

9.4 Visualisations as game transition systems

The idea of this work is to view visualisations as game transition systems, synchronised with formal models. 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 and changes to what can be seen can be considered as transitions. As an example, consider Fig. 9.3. The semi-circles represent states of the visualisation and the rectangles are the labelled transitions leading from one state to another. In the left semi-circle we see one state, a person is standing at the left of a line. If we take the transition in the leftmost rectangle, the person runs to the right and we reach the state in the semi-circle in the middle of the figure, where the person is standing at the middle of the line. Now one of two things can happen: either the person keeps running (the transition to the upper right state), or the person gets tired and sits down (the transition to the lower right state). This visualisation is a renaming of the TS in Fig. 9.2(a), where "runs" corresponds to transition a, "gets tired and rests" corresponds to transition b, and $s1 \ldots s4$ corresponds to the various positions of the person. The states are graphical images and the transitions are transformations of one graphical image to another, e.g., an animation.

If we allow all synchronisation 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, so if we, e.g., create a visualisation consisting of only one state and no transitions, the synchronisation is also without behaviour, which is not what we want to obtain. We want the behaviour of the synchronised system to be dictated by the model, and will only use the visualisation to show what happens in the model. In order to do this, we require that the visualisation is able to simulate the model. In that way, the behaviour of the synchronisation is dictated entirely by the model. A slight technicality is that the definition of a simulation (Def. 9.3) requires that the two systems share transitions. We remove this requirement and only require that, given a synchronisation constraint $\mathcal{S} \subseteq T_1 \times T_2$, whenever $s_1 \preceq s_2$, then for all $\alpha \in T_1$ if $s_1 \xrightarrow{\alpha} s_1'$ there exists a $s_2' \in S_2$ and a $\beta \in T_2$ such

that $s_1' \preceq s_2'$, $(\alpha, \beta) \in \mathcal{S}$, and $s_2 \xrightarrow{\beta} s_2'$. This allows us to say that (c) in Fig. 9.2 can be simulated by (a), (b), and (d), as we no longer care about the exact names of the transitions. For example, (c) can be simulated by (a) using the synchronisation constraint $\mathcal{S} = \{((a,a),a),((b,b),b)\}$ and the simulation $\preceq_b = \{((s1,s1),s1),((s2,s2),s2),((s3,s4),s3)\}$.

If we synchronise a model with a visualisation and require that the visualisation is able to simulate the model, the execution is defined by the model alone, which is fine if we only want to see the execution of the model. If we also want to manipulate the execution, 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 corresponds to 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. 9.6.

Definition 9.6 (Visualisation) *Given a model as a game $\mathcal{TS}_M = (S_M, T_M^u, T_M^c, \delta_M, s_{IM}, W_M)$, a **visualisation** $\mathcal{TS}_V = (S_V, T_V^u, T_V^c, \delta_V, s_{IV}, W_V)$, 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 $\mathcal{TS}_V$ can be used as a visualisation of $\mathcal{TS}_M$ 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 $s_{IM} \sim s_{IV}$.

The definition captures the intuition that whenever the model makes a move (a controllable transition in the model), the visualisation must be able to show that, and whenever the user provides some stimulation (a controllable transition in the visualisation), the model must be able to handle that and execute a corresponding uncontrollable transition.

One way to generate simple visualisations, is to use other formalisms as visualisation of our model. If we have created a model as a TS and need to communicate the model to an engineer who does not understand it, but who uses message sequence charts (MSC) on a daily basis (MSC can be seen as simplified UML sequence diagrams [131]), we can simply create an MSC and use it as visualisation of our model. In the following we present two visualisations that have proven themselves widely applicable and useful [T4, 94] for describing complex systems to domain experts, namely message sequence charts and cartoon-like visualisations created using a Java library called SceneBeans [149]. The MSC visualisation is an instance of the idea of using another formalism as visualisation of the model. The SceneBeans library is used by the LTSA tool as described in Sect. 9.2, but we use it in a way that makes it usable for a much wider range of formal models, as we do not require that the model is described as timed transition systems. The MSC visualisation exemplifies how to construct a visualisation that allows us to only see the behaviour of a model (not to manipulate it), whereas the SceneBeans visualisation allows us to see and manipulate the behaviour of the system.

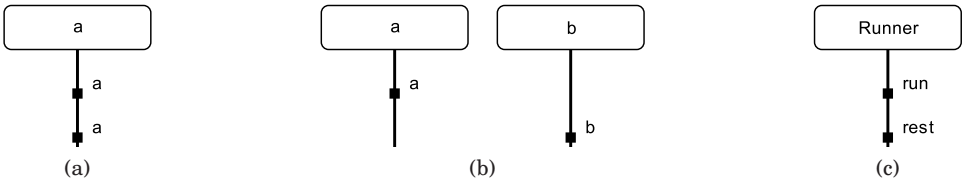


Figure 9.4: Simple MSC visualisations of the example from Fig. 9.2(a).

Example 1: Message sequence charts

Message sequence charts can be used either formally or informally to describe the behaviour of systems. A MSC consists of a set of processes, shown as vertical lines, which are able to exchange messages, represented as horizontal arrows from the source of the message to the destination, or which can execute internal events, represented as a dot on the process.

In its simplest form, this visualisation has a process for each transition and shows an internal event on the corresponding process whenever a transition is executed in the model. More formally, given a model $\mathcal{TS}_M = (S, \emptyset, T, \delta, s_I, W)$, we define a visualisation $\mathcal{TS}_V = (S_V, T_V, \emptyset, \delta_V, s_{IV}, \emptyset)$, where $T_V = T$. The set of states, S_V consists of all possible message sequence charts with T as processes. This is of course not manageable in reality (T may be infinite), so in practise we create processes as transitions are executed. The initial state is a MSC with processes T and no events, and transitions are enabled in $s \in S_V$, $s \xrightarrow{t} s'$, if $s' \in S_V$ is equal to s with an internal event added to the process t. The synchronisation used is equality. Using this visualisation directly on the model in Fig. 9.2(a), we can obtain the two leftmost visualisations in Fig. 9.4, the leftmost MSC, (a), corresponds to the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{a} s4$ and the middle MSC, (b), corresponds to the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$. The MSCs are updated as the model is executed, and the versions shown here are snapshots when no more transitions are enabled.

To make the visualisation more useful, we parametrise it with a function mapping transitions to process names and event labels so we can rename events and show "similar" events on a single process. Say the TS in Fig. 9.2(a) models a runner on a track (like the system in Fig. 9.3). The runner starts at the beginning of the track and runs towards the end. Optionally, the runner refuses to run any further halfway through the track, but sits down and rests. If we map the transition a to the process "Runner" and the event label "run" and b to "Runner" and "rest", we would obtain a visualisation as shown in Fig. 9.4(c) for the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$. This visualisation makes it easier to see what was intended by the model than the ones in Figs. 9.4(a) and (b).

Synchronising visualisations with formal models using this technique is very useful and allows us to observe what happens in the model, but it does not allow us to interact with the model, e.g., to drive the model into states we find interesting. The next example makes full use of the separation of transitions into controllable and uncontrollable transitions, and allows the user to interact with the model using the visualisation.

Example 2: Visualisation using SceneBeans

The SceneBeans [149] library uses an XML specification for describing visualisations and allows programs using it to interact with the visualisation by

invoking commands in the visualisation and receiving events from the visualisation. By invoking a command on a SceneBeans visualisation, it is possible to change what is displayed on the screen, e.g. to move a graphical representation of a person, as in Fig. 9.3, and thereby provide feedback to the user. When a user, e.g., clicks on an object in a visualisation, the visualisation can raise an event, which can be handled by the application. We equate uncontrollable transitions of a SceneBeans visualisation with the provided commands, and controllable transitions with the events that can be raised by user interaction.

Using the SceneBeans library, it is possible to create a visualisation like the one sketched in Fig. 9.3. We want to control the runner, so we switch the transitions in Fig. 9.2(a), so the dashed arcs represent controllable transitions and solid arcs represent uncontrollable transitions. The start of the track corresponds to the left of the line and the end of the track is at the right. When we want the runner to progress along the track, we can click the figure representing the runner to raise a "run"-event (corresponding to a controllable a transition in the TS in Fig. 9.2(a)). If we do nothing when the runner is halfway through the track (the middle state in Fig. 9.3), it is possible that the uncontrollable b transition is executed, leading to executing of the command "rest", which makes the runner sit down and rest.

This kind of visualisation is formalism-independent, but the visualisation is heavily dependent on the model, as we need to support the required commands and events. Furthermore the user is required to specify how events and commands should be synchronised with the transitions of the model.

9.4.1 Tool support

Support for synchronising visualisations with formal models by regarding the visualisations as games has been added to the BRITNeY Suite [C2, T3], a tool for visualising formal models, typically created using coloured Petri nets.

The tool has been extended with an interface, written in Java, which gives developers the ability to write their own programs interfacing with formal models. The interface, which can be seen in Fig. 9.5, informs a visualisation, i.e. a class implementing the interface, of all enabled controllable transitions (line 2). The visualisation returns which controllable moves it would like to perform. The visualisation is informed whenever the computer makes a move (line 4) and when a user-specified move is executed (line 3). The names controllable/uncontrollable are from the point of view of the visualisation. The tool is able to switch controllable/uncontrollable transitions so the visualisation can control the controllable transitions of the model if we wish to experiment with the behaviour of the model, or control the uncontrollable transitions of the model, allowing us to see how the model reacts to the surroundings. Finally, the visualisation is informed when there are no more enabled transitions (the game is over, line 5). This can be used if the user should be alerted or cleanup is needed when the game is over. Classes implementing this interface act as both visualisation and synchronisation constraint. As uncontrollable is not allowed to raise exceptions, visualisations implementing this interface are able to execute a transition synchronised with any transition offered by the model, so the visualisation's uncontrollable transitions are able to simulate the model's controllable transitions. Additionally, if controllable (in line 2) returns a subset of the transitions provided as parameter, the model is able to simulate the controllable transitions of the visualisation. Thus, according to Def. 9.6, a class implementing the interface in Fig. 9.5 can be used as visualisation of any model.

```
public interface GameListener {
      List<Transition> controllable(List<Transition> ts);
      void controllable(Transition t);
      void uncontrollable(Transition t);
      void gameOver();
}
```

Figure 9.5: The GameListener interface.

Both of the visualisations presented in this section have been implemented using this interface, so despite the simplicity of the interface, it is versatile. In addition to the examples in this paper, the interface has also been used to implement a visualisation which automatically generates form-filling dialogues for CPN models (this visualisation is described in [C4]) as well as for ongoing work on implementing a work-flow system on top of game CP-nets.

Fairness

If the purpose of a visualisation is to get acquainted with the model or the modelled system, it is often reasonable to assume that a computer tool chooses controllable transitions at random. This can often be done very quickly, however, and this can make it difficult for the user to interact with the model. To overcome this, we may need to impose fairness during execution of the model.

A simple way to impose fairness is to make the game turn-based: the model makes one uncontrollable transition, followed by a user-selected controllable transition and so on until no transitions are enabled. If either of the players have no possible moves (i.e. no controllable resp. uncontrollable transitions are enabled) the turn is passed on to the other player. This approach is simple, easy to understand, and easy to implement. The disadvantage is that, depending on modelling detail, one player may gain an unfair advantage, and minor changes may make it difficult for one player to keep up with the moves of the other. This can be seen in the runner example in Fig. 9.2(a) (with $T^c = \{b\}$ and $T^u = \{a\}$), where we always end up in state $s3$ if we use this technique, as we need to first choose an a transition. The turn is then passed on to the computer, which chooses a b transition.

Another way to impose fairness is to give controllable transitions (from the visualisation's point of view) priority over uncontrollable transitions. This particularly useful for SceneBeans visualisations, where transitions of the model are expected to be executed while the user is observing, but we want interaction to happen immediately when the user requests it. The visualisation of the runner will do nothing until a controllable transition has been chosen in the runner example in Fig. 9.2(a) and Fig. 9.3. When the runner is in the state $s2$, at the middle of the track, a user can force the runner along the track by clicking on the graphical representation of the runner. If the user does nothing for a while, the computer will choose the uncontrollable transition b, and the runner will rest.

A third way to impose fairness is to make execution of transitions take time. A simple way to do this is to let the execution of every transition take, say, 0.1 second. The advantages and disadvantages of this approach is the same as those for turn-based execution. A slightly more involved way to use time to impose fairness is to use a timed formalism such as timed automata [1]. In this case, transitions may only be enabled for a certain amount of model-time

or only a certain amount of model-time after another transition. The idea is to let model-time correspond to real time. The advantage of this approach is that it is very general, and allows us to get a natural feeling of the behaviour of a timed model, but the disadvantage is that timed models may be more difficult to understand and this approach requires a timed formalism.

9.5 Use of visualisations

In this section we give two examples of use of visualisations. The first example in an industrial case study, where visualisation is used to improve a specification, using the methodology in Fig. 9.1, and the second example is to an application to verification of games, where we use visualisations to convince domain experts that no winning strategy exists (when it is believed that it should) as well as providing a means to find out if the error is in the specification or the formal model.

9.5.1 Industrial Case: Routing in Mobile Ad-hoc Networks

First, we look at an industrial application of visualisation, which uses an earlier version of the BRITNeY Suite without support for visualisations as games. The project is a collaboration between Ericsson Denmark A/S, Telebit and the CPN group at the University of Aarhus. For more details about the project, see [T4].

In Fig. 9.6, we see two visualisations created to visualise an interoperability protocol for mobile ad-hoc networks. The protocol ensures that mobile ad-hoc nodes (laptops) can communicate with a stationary host when on the move via the nearest gateway. 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 gateway to use. The basic operation of the model is illustrated by the MSC in Fig. 9.6 (top). The protocol is modelled using coloured Petri nets in a model that contains modules, 54 places and 40 transitions. Altogether the model also contains 1000 lines of inscriptions, 200 of which are used to drive the visualisation. The exact details of the protocol are out of scope of this paper. The visualisation in Fig. 9.6 (bottom) makes it possible for the user to observe the behavior of the system as packets, visualised by colored dots, flow along the network and to provide stimuli to the protocol by dragging and dropping the laptops to indicate the node movement. These visualisations have been used in the project, both internally during protocol design, and externally, when presenting the protocol to management and protocol engineers unfamiliar with formal modeling.

The project uses the visualisations described in Sect. 9.4, namely a message sequence chart and a SceneBeans visualisation. The visualisations have been synchronised with the model using annotations of the model. One of the problems we encountered during the project, was this need to add annotations to the model. For example, in Fig. 9.7, we see the annotation “input...”, used to show packets flow. This is by far the largest annotation of the model, and clutters it unnecessarily. Furthermore annotations have to be added for each visualisation, making it difficult to turn off one or more visualisations, in order to focus on e.g. the MSCs. Using the approach described in this paper, we create our visualisations and for each specify how it should be synchronised with the model (in fact we would not need to specify synchronisation constraints as the implementation uses conventions, such as naming, to generate these automatically), and we can then turn off each visualisation independently and

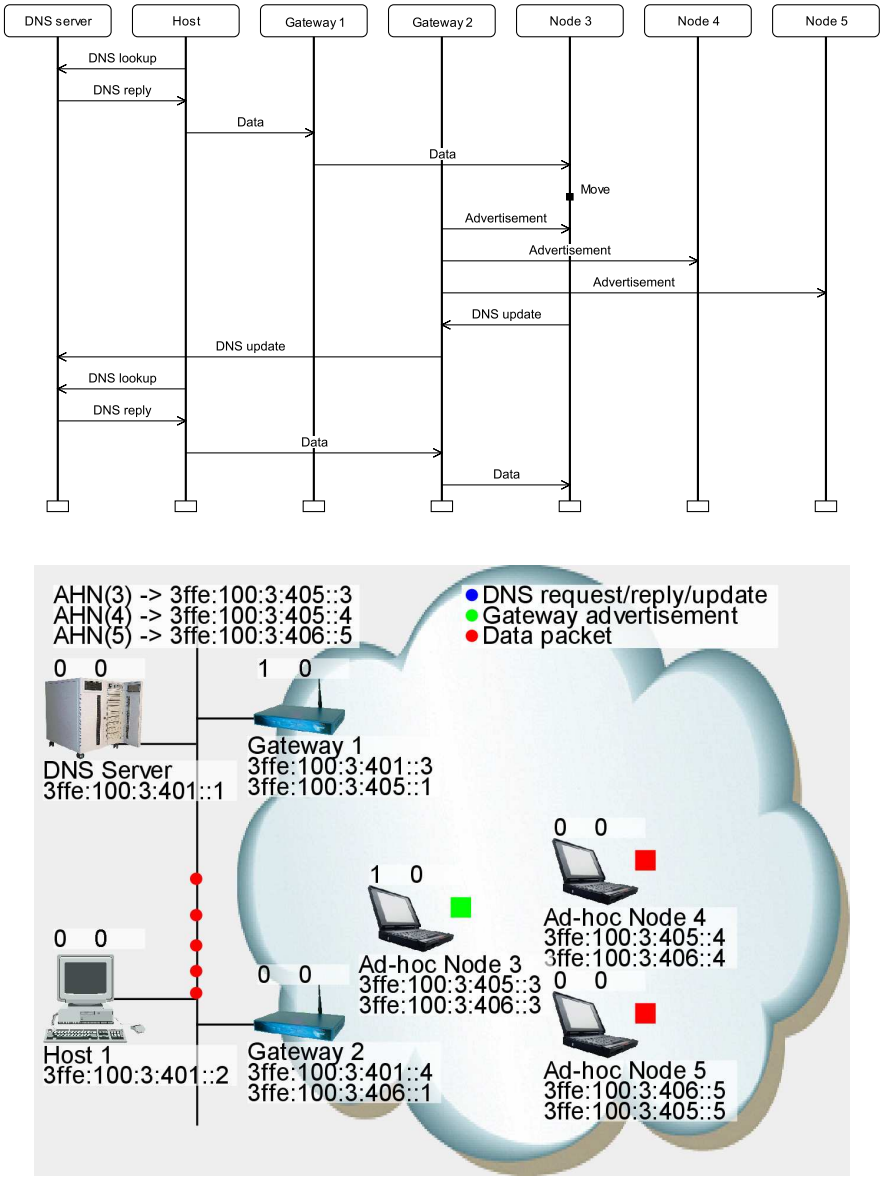


Figure 9.6: Visualisations used in an industrial project.

the model would be left uncluttered. Another major problem encountered in the project was that we wanted the model to perform actions when idle, e.g. send gateway advertisements, and react immediately when we moved a node or wanted to send packets. We only partially solved this by polling the visualisation for changes, which made the visualisation almost work, but was never satisfactory. Creating the visualisations as games, as proposed in this paper, making slight changes to the model in order to make it a game (make e.g. the movement of the ad-hoc nodes uncontrollable in the model), and using one of the fairness constraints discussed in Sect. 9.4.1, it is possible to make interaction with the model much more natural as the visualisation will we able to force actions in the formal model as desired.

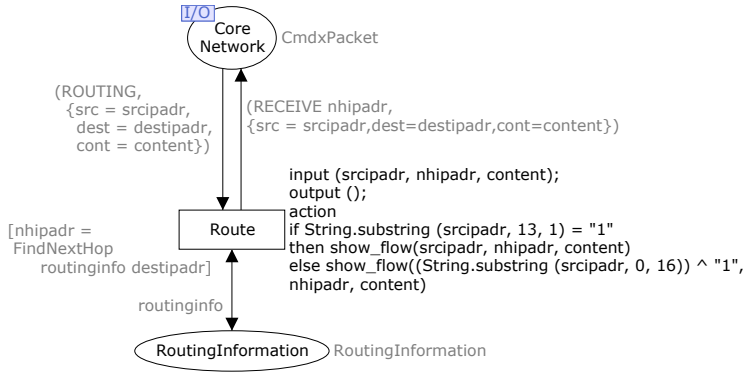


Figure 9.7: Part of the routing mechanism.

9.5.2 Visualising winning strategies

Hitherto, we have used visualisations primarily for validation that the formal model reflects the intended behaviour by letting a domain expert stimulate and observe the model using visualisations. Now, we will turn to using visualisation for communicating the result of formal verification, i.e., convincing users that no winning strategy exists, which is decided using an algorithm from [18] as outlined in Sect. 9.3. The purpose of a counter example is to convince users that it is impossible to have a winning strategy, so we let the domain expert assume the role of the modelled system and let him try out ideas for winning strategies. At the same time we let a computer tool take charge of the uncontrollable actions according to the counter example that has been calculated. The user is urged to reach a winning state while the tool executes uncontrollable transitions to prevent that (by ensuring that the user is not allowed the ability to execute a transition leading to a good state). We can do this using the formal model, but often the formal methods expert does not have enough domain knowledge to have understand why the system should have winning strategy, so the domain expert, who has little knowledge of the modelling language, has to find out whether the error is in the model or in the specification. Instead we let the domain expert control the controllable transitions of the model using a visualisation (the computer tool is able to let the visualisation assume control of either the controllable or uncontrollable transitions, as described in Sect. 9.4.1). We let the user stimulate the model in any way seen fit (according to the supposed winning strategy), and eventually the model will perform an unforeseen move (error in the specification) or the model will perform a disallowed move (error in the model).

In the example in Fig. 9.2(a), we may think we have a winning strategy: always pick transition a. This leads to the winning state $s4$, right? The computer tool knows that the only good state is $s4$, and will stay clear of it. If we let the user use the visualisation in Fig. 9.3, he would first click the runner to progress to $s2$. The tool then executes the b transition as it knows that $s4$ is good, but $s3$ is not. The game is over and the user is hopefully convinced that it is impossible to ensure we end up in a winning state.

9.6 Conclusion and future work

In this paper we have given a theoretical foundation for viewing visualisations as game transition systems synchronised with formal models, providing

a uniform and general framework for coupling formal models and behavioural visualisation. We have used game-theory to separate output from and input to the model and given two concrete examples of visualisations. We have demonstrated how an industrial case can benefit from using the method described in this paper. Furthermore, we have sketched how this can be used to create counter-examples to the existence of a winning strategy in games, so domain experts with no knowledge of the formalism used can understand them.

Future work includes using this technique in industrial settings. The visualisations described in this paper is already distributed as part of the BRITNeY Suite, and ongoing work on creating a detailed model of TCP/IP uses the MSC visualisation to communicate the model to protocol experts.

Index

Bibliography

Papers part of this thesis

[T1] T. Mailund and M. Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 177–191. Springer-Verlag, 2004.

[T2] M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *Proc. of ATPN'07*, volume 4546 of *LNCS*, pages 446–464. Springer-Verlag, 2007.

[T3] M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.

[T4] L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.

[T5] M. Westergaard. A Game-theoretic Approach to Behavioural Visualisation. Submitted, 2007.

Papers co-authored by the author of this thesis

[C1] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.

[C2] M. Westergaard. BRITNeY suite website. Online: `wiki.daimi.au.dk/britney/`.

[C3] M. Westergaard. Building Verifiable Software Prototypes using Coloured Petri Nets. Progress report, Department of Computer Science, University of Aarhus.

[C4] M. Westergaard. Game Coloured Petri Nets. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 281–300, 2006.

[C5] M. Westergaard. The BRITNeY Suite: A Platform for Experiments. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 97–116, 2006.

[C6] M. Westergaard and K.B. Lassen. Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY animation and CPN Tools. In *Proc. of 6th CPN Workshop*, volume 576 of *DAIMI-PB*, pages 119–136, 2005.

Other references

[1] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] A. Arnold. *Finite transition systems. Semantics of communicating systems*. Prentice-Hall, 1994.

[3] ASCoVeCo Project webpage. Online: `www.daimi.au.dk/~ascoveco/`.

[4] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of POPL'02*, pages 1–3. ACM Press, 2002.

[5] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[6] G. Behrmann, K.G. Larsen, and R. Pelánek. To Store or Not to Store. In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 433–445. Springer-Verlag, 2003.

[7] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer-Verlag, 2004.

[8] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking withoud BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[9] J. Billington, M.C. Wilbur-Ham, and M.Y. Bearman. Automated protocol Verification. In *Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification*, pages 59–70. Elsevier, 1985.

[10] C. Bossen and J.B. Jørgensen. Context-descriptive prototypes and their application to medicine administration. In *Proc. of DIS'04*, pages 297–306. ACM Press, 2004.

[11] W.S. Brainerd and L.H. Landweber. *Theory of Computation*. John Wiley & Sons, Inc., 1974.

[12] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[14] C. Capellmann, S. Christensen, and U. Herzog. Visualising the Behaviour of Intelligent Networks. In *Services and Visualisation, Towards User-Friendly Design*, volume 1385 of *LNCS*, pages 174–189. Springer-Verlag, 1998.

[15] L. Cardelli, G. Ghelli, and A.D. Gordon. Types for the Ambient Calculus. To Appear in I&C special issue on TCS'2000, 2001.

[16] L. Cardelli and A.D. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *Proc. of POPL'00*, pages 365–377. ACM Press, 2000.

[17] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[18] F. Cassez, A. David, F. Emmanuel, K.G. Larsen, and D. Lime. Efficient On-the-fly Algorithms for the Analysis of Timed Games. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer-Verlag, 2005.

[19] T. Cattel. Modelling and Verification of sC++ Applications. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 232–248. Springer-Verlag, 1998.

[20] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Online: `www.w3.org/TR/wsdl20/`.

[21] G. Chiola, S. Donatelli, and G. Franceschinis. Priorities, Inhibitor Arcs and Concurrency in P/T nets. In *Proc. of ATPN'91*, pages 182–205, 1991.

[22] S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Proc. of ATPN'94*, volume 815 of *LNCS*, pages 159–178. Springer-Verlag, 1994.

[23] S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN—A Computer Tool for Coloured Petri Nets. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.

[24] S. Christensen and L. M. Kristensen. State Space Analysis of Hierarchical Coloured Petri Nets. *Petri Net Approaches for Modelling and Validation, Lincom Studies in Computer Science 01*, pages 1–16, 2003.

[25] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.

[26] S. Christensen, L.M. Kristensen, and T. Mailund. Condensed State Spaces for Timed Petri Nets. In *Proc. of ATPN'01*, volume 2075 of *LNCS*, pages 101–120. Springer-Verlag, 2001.

[27] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[28] E. M. Clarke, O. Grumberg, M. Minna, and D. Peled. State Space Reduction using Partial Order Techniques. *Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[29] T. Clausen and P. Jacquet. Optimised Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[30] CMMI Product Team. CMMI for Development, Version 1.2. Technical report, Carnegie Mellon University, 2006. CMU/SEI-2006-TR-008; online version: `www.sei.cmu.edu/publications/documents/06.reports/06tr008.html`.

[31] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[32] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.

[33] CPN Tools webpage. Online: `www.daimi.au.dk/CPNTools/`.

[34] The CPN Group at University of Aarhus. Online: `www.daimi.au.dk/CPnets`.

[35] APN ML Protocol Manual. Online: `wiki.daimi.au.dk/cpn2000/apn_ml_protocol_manual.wiki`.

[36] J. Desel and W. Reisig. Place/Transition Petri Nets. In *Lectures on Petri nets I: Basic Models*, volume 1491 of *LNCS*, pages 122–173. Springer-Verlag, 1998.

[37] Design/CPN website. Online: `www.daimi.au.dk/designCPN/`.

[38] P.C. Dillinger and P. Manolios. Fast and accurate Bitstate Verification for SPIN. In *Proc. of SPIN 2004*, volume 2989 of *LNCS*. Springer-Verlag, 2004.

[39] M. Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997.

[40] M. Dwyer, J. Hatclif, V. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reductions for Concurrent Object-Oriented Programs, 2004. Formal Methods in System Designs. (to appear).

[41] Eclipse webpage. Online: `www.eclipse.org/`.

[42] S. Edelkamp, A.L. Lafuente, and S. Leue. Directed Explicit Model Checking with HFS-SPIN. In *Proc. of SPIN'01*, volume 2057 of *LNCS*, pages 57–79. Springer-Verlag, 2001.

[43] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. of SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer-Verlag, 2003.

[44] P. Ehn and M. Kyng. Cardboard Computers: Mocking-it-up or Hands-on the Future. In *Design at Work: Cooperative Design of Computer Systems*, pages 169–196. Lawrence Erlbaum Associates, Inc., 1992.

[45] I.K. El-Far and J.A. Whittaker. Model-based Software Testing. *Encyclopedia of Software Engineering*, 1:825–837, 2002.

[46] E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996.

[47] Ericsson Danmark A/S, Telebit. Online: `www.tbit.dk`; no longer available as the company has been taken over, announcement in danish at `www.tietoenator.dk/default.asp?path=487,612,26101,26490`.

[48] E.E. Euler, S.D. Jolly, and H.H. Curtis. The failures of the Mars Climate Orbiter and Mars Polar Lander: A perspective from the people involved. In *Proc. of Guidance and Control 2001*. American Astronautical Society, 2001.

[49] S. Evangelista and J.-F. Pradat-Peyre. Memory Efficient State Space Storage in Explicit Software Model Checking. In *Proc. of SPIN'05*, volume 3639 of *LNCS*, pages 43–57. Springer-Verlag, 2005.

[50] The ExSpect tool webpage. Online: `www.exspect.com`.

[51] The FeaVer Feature Verification System webpage. Online: `cm.bell-labs.com/cm/cs/what/feaver/`.

[52] A. Finkel. A Minimal Coverability Graph for Petri Nets. In *Proc. of ATPN'90*, pages 1–21, 1990.

[53] G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of Third CPN Workshop*, volume 554 of *DAIMI-PB*, pages 79–93, 2001.

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

[55] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[56] J. Geldenhuys. State Caching Reconsidered. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 23–38. Springer-Verlag, 2004.

[57] J. Geldenhuys and A. Valmari. A Nearly Memory-Optimal Data Structure for Sets and Mappings. In *Proc. of SPIN 2003*, volume 2648 of *LNCS*, pages 136–150. Springer-Verlag, 2003.

[58] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proc. of CAV'90*, volume 531 of *LNCS*, pages 175–186. Springer-Verlag, 1990.

[59] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.

[60] P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.

[61] S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.

[62] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition)*. The Java Series. Prentice Hall, 2005.

[63] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau H.F Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Online: `www.w3.org/TR/soap12-part1/`.

[64] B. Han and J. Billington. Formalising the TCP Symmetrical Connection Management Service. In *Proc. of DASD'03*, pages 178–184. SCS, 2003.

[65] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[66] D. Harel and R. Marelly. *Come, Let's Play*. Springer-Verlag, 2003.

[67] D. Harel and P.S. Thiagarajan. Message Sequence Charts. In *UML for Real: Design of Embedded Real-Time Systems*, pages 77–105. Kluver Academic Publishers, 2003.

[68] R. Hauser and J. Koehler. Compiling Process Graphs into Executable Code. In *Proc. of GPCE'04*, volume 3286 of *LNCS*, pages 317–336. Springer-Verlag, 2004.

[69] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513, April 2003.

[70] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[71] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[72] G.J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software, Practice and Experience*, 18(2):137–161, 1988.

[73] G.J. Holzmann. Algorithms for Automated Protocol Validation. *AT&T Technical Journal*, 69(2):32–44, 1990.

[74] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.

[75] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Trianing Runs. In *Proc. of SPIN'97*, 1997.

[76] G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.

[77] G.J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[78] G.J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. *Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[79] G.J. Holzmann and M.H. Smith. A practical method for verifying event-driven software. In *Proc. of ICSE'99*, pages 597–607. IEEE Comp. Soc. Press, 1999.

[80] G.J. Holzmann and M.H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[81] T. Huckle. Collection of software bugs. Online: `www5.in.tum.de/~huckle/bugse.html`, February 28 2007.

[82] C. Huitema. *IPv6: The New Internet Protocol*. Prentice-Hall, 1998.

[83] HyperSAT website. Online: `www.cs.ubc.ca/~babic/index_hypersat.htm`.

[84] Standard for Modeling and Simulation High Level Architecture. IEEE standard 1516.

[85] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9, 1996.

[86] Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding. ISO/IEC 14772-1:1997.

[87] Software and system engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation. ISO/IEC 15909-1:2004.

[88] JavaBeans(TM) Specification 1.01 Final Release. Online: `java.sun.com/products/javabeans/docs/spec.html`.

[89] Java Plug-in Framework website. Online: `jpf.sourceforge.net/`.

[90] JBuilder website. Online: `www.codegear.com/Products/JBuilder`.

[91] K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.

[92] K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. Springer-Verlag, 1994.

[93] K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.

[94] J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA'05*, 2005.

[95] JSR 56: Java Network Launching Protocol and API. Online: `jcp.org/en/jsr/detail?id=56`.

[96] T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.

[97] R.M. Karp and R.E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 4:147–195, 1969.

[98] B.W. Kernigan and D.M Ritchie. *The C Programming Language (2nd Edition)*. Prentice-Hall, 1988.

[99] E. Kindler and C. Páles. 3D-Visualization of Petri Net Models: Concept and Realization. In *Proc. of ATPN'04*, volume 3099 of *LNCS*, pages 464–473. Springer-Verlag, 2003.

[100] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-view-controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Computing*, 1(3):26–49, August/September 1988.

[101] L.M. Kristensen. Ad-hoc Networking and IPv6: Modelling and Validation. Online: `www.pervasive.dk/projects/IPv6/IPv6_summary`.

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

[103] L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.

[104] L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.

[105] L.M. Kristensen and T. Mailund. Path Finding with the Sweep-Line Method using External Storage. In *Proc. of FEM'03*, volume 2885 of *LNCS*, pages 319–337. Springer-Verlag, 2003.

[106] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2): 312–360, 2000.

[107] K.G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proc. of EMSOFT'05*, pages 299–306. ACM Press, 2005.

[108] X. Leroy. *The Objective Caml system release 3.10: Documentation and user's manual*. Online: `caml.inria.fr/pub/docs/manual-ocaml/`, 2007.

[109] N.G. Leveson and C.S. Turner. Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[110] A. Lindem. OSPF for IPv6. Internet-draft, March 2005.

[111] X. Liu and S.A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In *Proc. of ICALP'98*, volume 1443 of *LNCS*, pages 53–64. Springer-Verlag, 1998.

[112] L. Lorentsen, A-P Tuovinen, and J. Xu. Modelling Features and Feature Interactions of Nokia Mobile Phones Using Coloured Petri Nets. In *Proc. of ATPN'02*, volume 2360 of *LNCS*, pages 294–313. Springer-Verlag, 2002.

[113] B. Lindstrøm and L. Wells. Towards a Monitoring Framework for Discrete-Event System Simulations. In *Proc. of WODES'02*, pages 127–134. IEEE Comp. Soc. Press, 2002.

[114] R.J. Machado, K.B. Lassen, S. Oliveira, M. Couto, and P. Pinto. Execution of UML Models with CPN Tools for Workflow Requirements Validation. In *Proc. of 6th CPN Workshop*, volume 576 of *DAIMI-PB*, pages 231–250, 2005.

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

[116] J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.

[117] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *Proc. of 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, 2000.

[118] T. Mailund. *Sweeping the State Space — A Sweep-Line State Space Exploration Method*. PhD thesis, Department of Computer Science, University of Aarhus, 2003.

[119] M. Mäkelä. Condensed Storage of Multi-Set Sequences. In *Proc. of Workshop on Practical Use of High-level Petri Nets*, volume 547 of *DAIMI-PB*, pages 111–126, 2000.

[120] J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.

[121] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[122] A.M. Mikkelsen. On-the-fly LTL Model Checking in Design/CPN. Master's thesis, Dept. of Computer Science, University of Aarhus, 2001.

[123] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[124] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[125] R. Milner. Bigraphical Reactive Systems. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*, pages 16–35. Springer-Verlag, 2001.

[126] R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.

[127] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[128] K.L. Morse, M. Lightner, R. Little, B. Lutz, and R. Scrudder. Enabling Simulation Interoperability. *Computer*, 39(1):115–117, 2006.

[129] Remote Sensing Satellites. Online: `earthobservatory.nasa.gov/Library/RemoteSensingAtmosphere/remote_sensing5.html`.

[130] P.C. Nørgaard. NCW Routing in Tactical Networks. Ericsson Danmark A/S, Telebit. Technical Report.

[131] Object Management Group. Unified Modeling Language (UML), Version 2.1.1. Online: `www.omg.org/technology/documents/formal/uml.htm`, 2007.

[132] C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *LNCS*, pages 292–302. Springer-Verlag, 2003.

[133] R. Pelánek. Typical Structural Properties of State Spaces. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 5–22. Springer-Verlag, 2004.

[134] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.

[135] D. Peled. Combining Partial Order Reductions with On-the-fly Model Checking. *Formal Methods in System Design*, 8:39–64, 1996.

[136] D. Peled. Ten Years of Partial Order Reduction. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 17–28. Springer-Verlag, 1998.

[137] C.E. Perkins. *Ad Hoc Networking*. Addison-Wesley, 2001.

[138] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, 1962. Schriften des IIM Nr. 2.

[139] C.A. Petri. *Interpretations of Net Theory*. St. Augustin: Gesellschaft für Mathematik und Datenverarbeitung Bonn, Interner Bericht ISF-75–07, Second Edition, 1976.

[140] Y.-M. Quemener and T. Jeron. Finitely Representing Infinite Reachability Graphs of CFSMs with Graph Grammars. In *Proc. of FORTE'96*, pages 364–379, 1996.

[141] J.L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. Online: `www.daimi.au.dk/designCPN`.

[142] J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ATPN'96*, volume 1091 of *LNCS*, pages 400–419. Springer-Verlag, 1996.

[143] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[144] J.B. Jørgensen, K.B. Lassen, and W.M.P. van der Aalst. From Task Descriptions via Coloured Petri Nets Towards an Implementation of a New Electronic Patient Record. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 17–35, 2006.

[145] Ó.R. Ribeiro and J.M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 237–256, 2006.

[146] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math Soc.*, 89:25–59, 1953.

[147] W.W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proc. of ICSE'87*, pages 328–338. Computer Society Press, 1987.

[148] Boolean satisfiability problem, June 29 2007. Wikipedia article: `en.wikipedia.org/wiki/Boolean_satisfiability_problem`.

[149] SceneBeans webpage. Online: `www-dse.doc.ic.ac.uk/Software/SceneBeans`.

[150] K. Schmidt. LoLA - A Low Level Analyser . In *Proc. of ATPN'00*, volume 1825 of *LNCS*, pages 465–474. Springer-Verlag, 2000.

[151] K. Schmidt. Using Petri Net Invariants in State Space Construction. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 473–488. Springer-Verlag, 2003.

[152] SLAM Project. Online: `research.microsoft.com/slam/`.

[153] S. Sørensen. Disk Based State Space Storage for Coloured Petri Nets. Master's thesis, Dept. of Computer Science, University of Aarhus, 2002.

[154] Spin Version 4: Language Reference. Online: `spinroot.com/spin/Man/promela.html`.

[155] U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.

[156] U. Stern and D.L. Dill. Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 172–183. Springer-Verlag, 1998.

[157] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[158] J. Toksvig. Design and Implementation of a Place Invariant Tool for Coloured Petri Nets. Master's thesis, Dept. of Computer Science, University of Aarhus, 1995.

[159] J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.

[160] A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets '90*, volume 483 of *LNCS*, pages 491–515. Springer-Verlag, 1990.

[161] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.

[162] W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements Via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In *Proc. of OTM Conferences (1)*, volume 3760 of *LNCS*, pages 22–39. Springer-Verlag, 2005.

[163] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[164] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[165] S. Vanit-Anunchai, J. Billington, and G.E. Gallasch. Sweep-line Analysis of DCCP Connection Management. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 157–176, 2006.

[166] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *In proc. of IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[167] Visual Studio website. Online: `msdn.microsoft.com/vstudio/`.

[168] P. Vixie. Dynamic Updates in the Domain Name System. RFC 2136, April 1997.

[169] M. Weber and E. Kindler. The Petri Net Kernel. In *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 109–123. Springer-Verlag, 2003.

[170] D. Winer. XML-RPC Specification. Online: `www.xmlrpc.org/spec`.

[171] P. Wolper and P. Godefroid. Partial Order Methods for Temporal Verification. In *Proc. of CONCUR'93*, volume 715 of *LNCS*. Springer-Verlag, 1993.

[172] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.