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

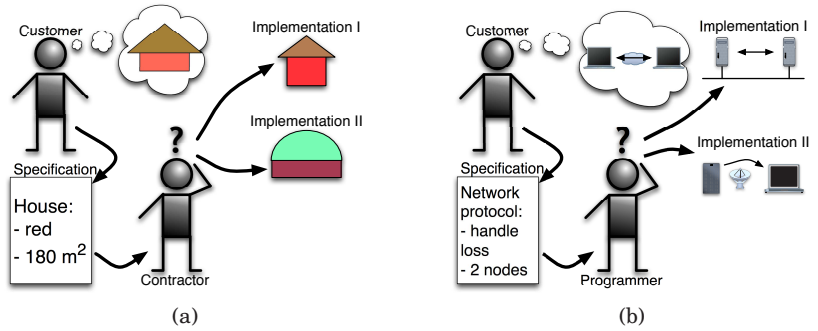
of formal models and how to visualise that behaviour. After that we provide directions for arriving at a correct implementation from the formal model, and we end this chapter by providing a guide to the rest of this thesis.

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.

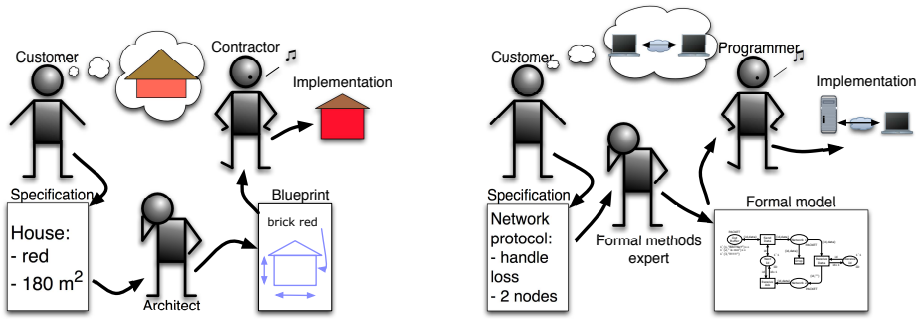
Given the limitation of classical approaches to correctness, we instead consider an approach based on construction of models. To illustrate this, we consider a parallel in the physical world, namely building a house. When a customer wants a new house, but is unable to build it himself, he hires some 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-



(a) (b)

Figure 1.1: Construction directly from the specification.



(a) (b)

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

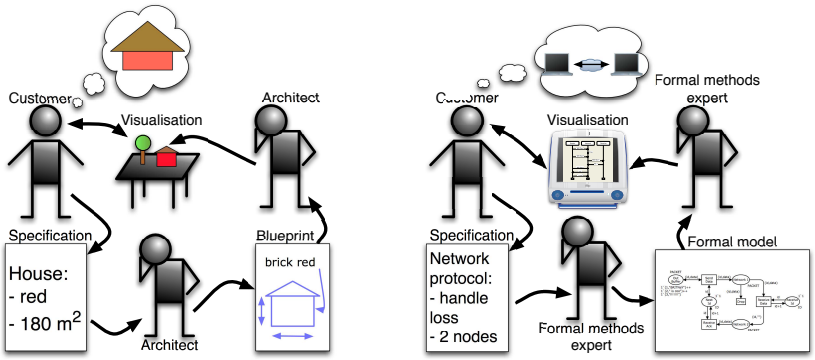


Figure 1.3: Using visualisation to ensure the formal model corresponds to the customers intentions.

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

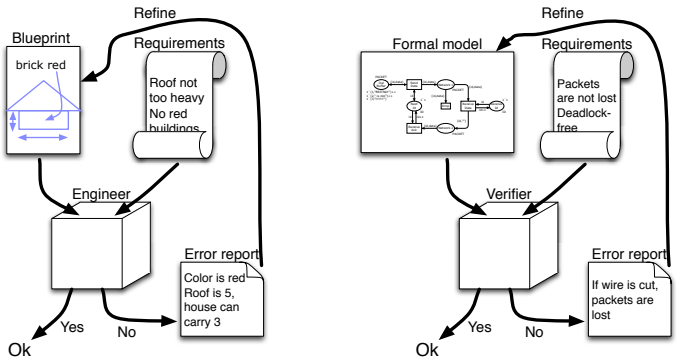


Figure 1.4: Verification of formal models.

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.

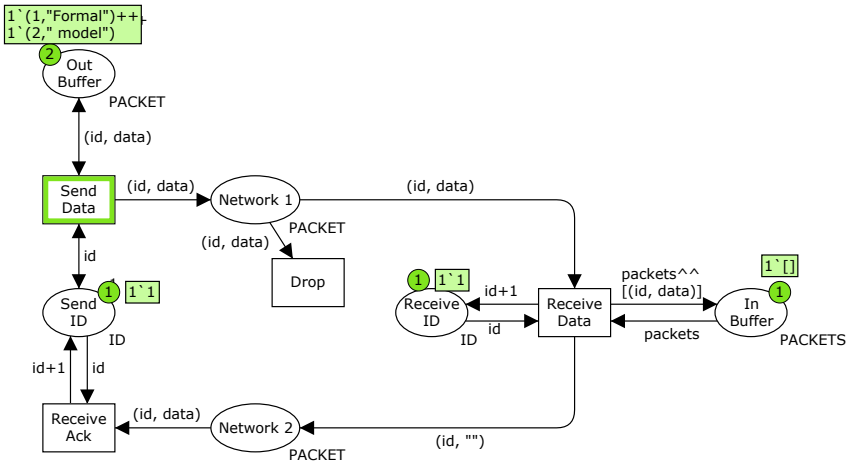
1.2 Behavioural Models of Concurrent Systems

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.

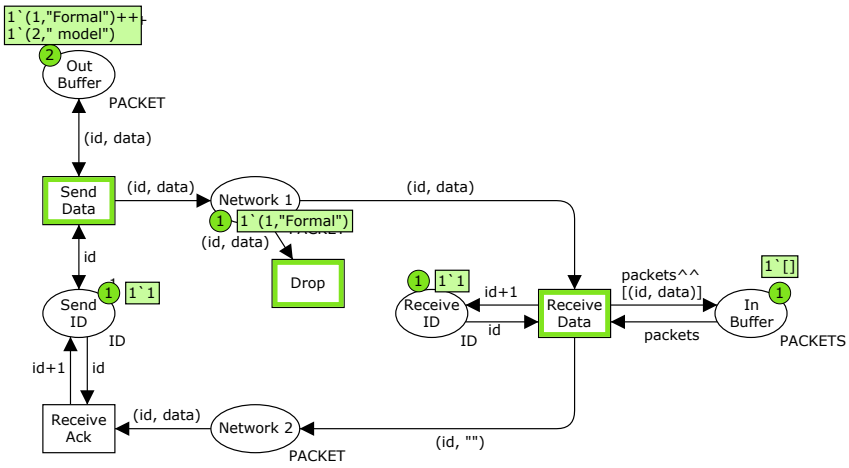
In the rest of this thesis we solely focus on models of the behaviour of concurrent systems [138], i.e., systems where computation is performed in multiple components or threads. Examples span from multi-threaded applications running on a single computer, such as a word processor which is able to continue 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

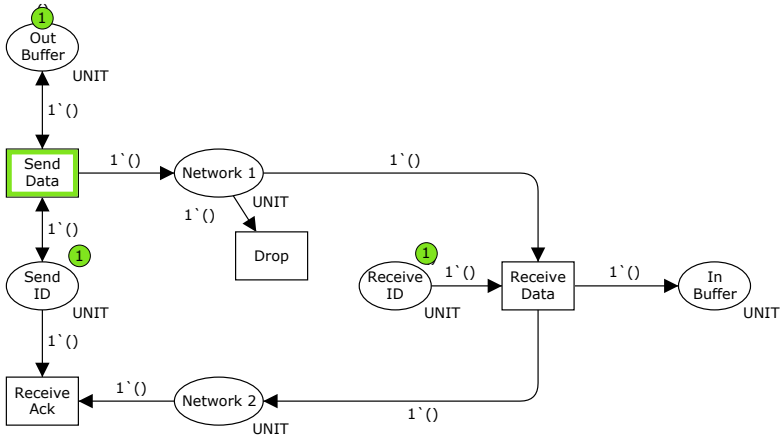


Figure 1.7: A simplified version of the network protocol from Fig. 1.7 created as a PT-net.

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

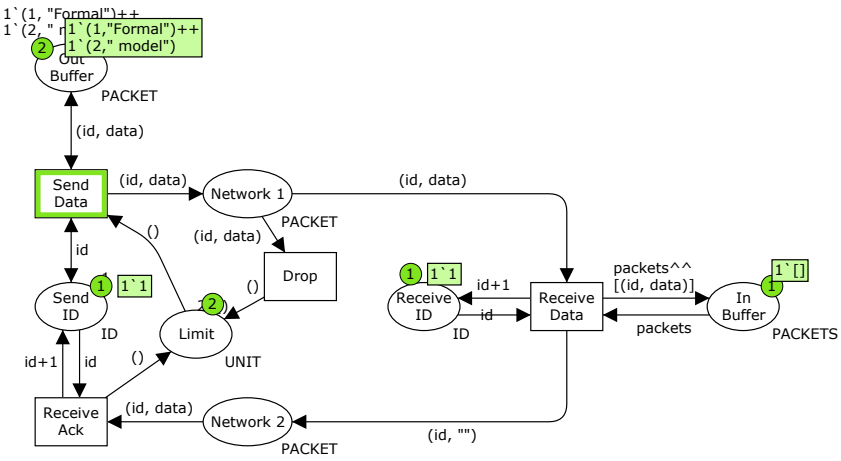


Figure 1.8: A version of the simple network protocol from Fig. 1.5 with a bound on the number of possible outstanding packets.

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

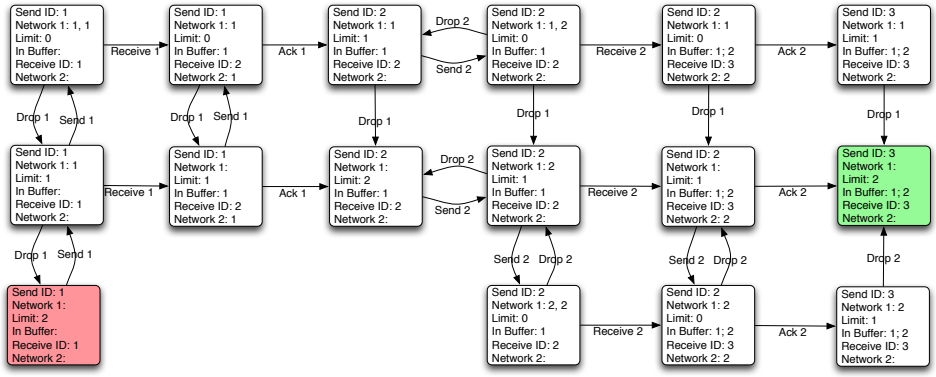


Figure 1.9: The reachability graph of the simple network protocol in Fig. 1.8.

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

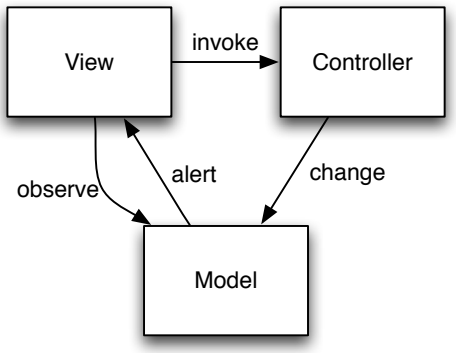


Figure 1.10: The Model-View-Controller design pattern.

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.

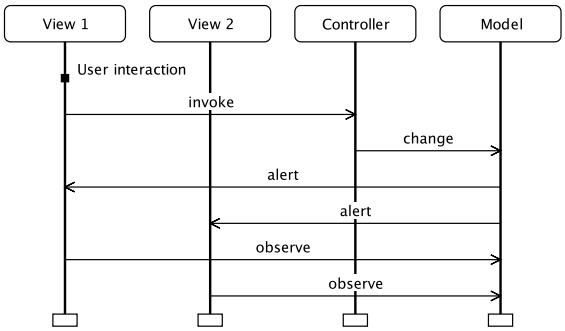


Figure 1.11: How two views associated with the same model are updated.

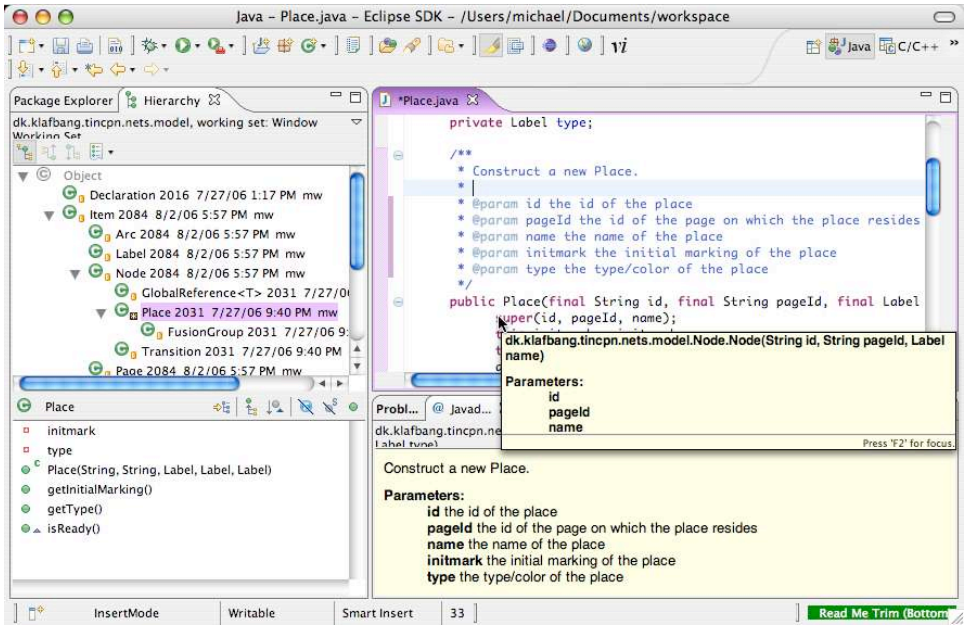


Figure 1.12: Screen-shot from the Eclipse Java editor.

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

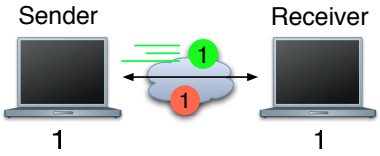


Figure 1.13: Visualisation of a simple network protocol.

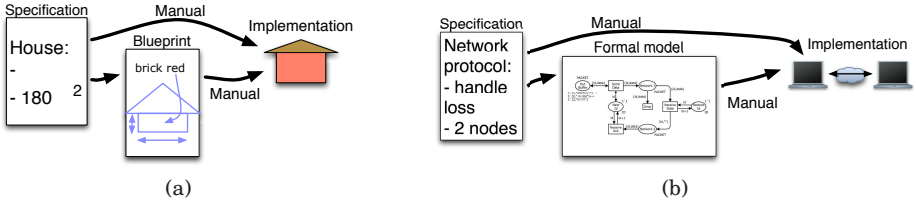


Figure 1.14: Manual construction.

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-

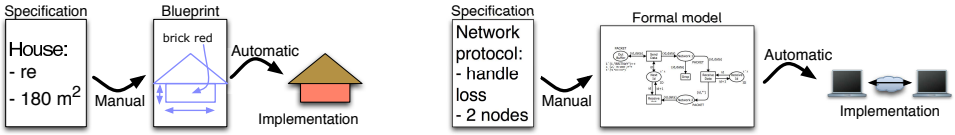


Figure 1.15: Synthesis.

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

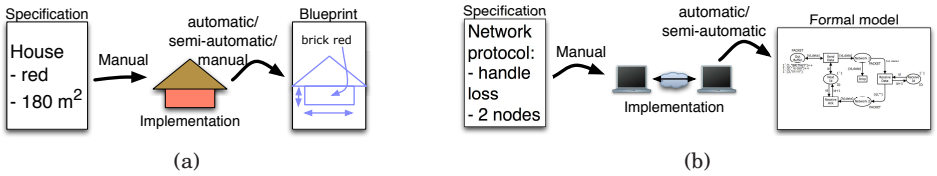


Figure 1.16: Testing using automatically generated formal models.

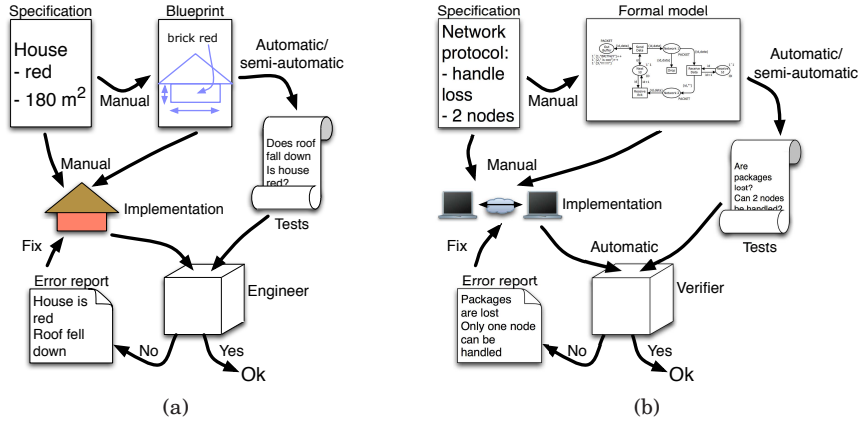


Figure 1.17: Using formal models to generate tests of the implementation.

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.