# Verification of interconnects

Joosten, S.J.C.

Published: 24/02/2016

*Document Version*
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

# Verification of Interconnects

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens,
voor een commissie aangewezen door het College voor Promoties,
in het openbaar te verdedigen op
woensdag 24 februari 2016 om 16:00 uur

door

Sebastiaan Jozef Christiaan Joosten

geboren te Enschede

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

| | |
|---|---|
| Voorzitter: | prof. dr. J. de Vlieg |
| Promotoren: | prof. dr. M. C. J. D. van Eekelen (Open Universiteit Nederland) |
| | prof. dr. ir. J. F. Groote |
| Copromotor: | dr. J. Schmaltz |
| Leden: | prof. dr. A. Biere (Johannes Kepler Universität Linz) |
| | prof. dr. K. G. W. Goossens |
| | prof. dr. T. F. Melham (University of Oxford) |
| | prof. dr. J. C. van de Pol (Universiteit Twente) |

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Sebastiaan Joosten

# Verification of Interconnects

Sebastiaan Joosten
Department of Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands

Cover: Bram Joosten

This research is supported by the NWO project Effective Layered Verification of Networks on Chip (ELVeN) under grant no. 612.001.108

A catalogue record is available from the Eindhoven University of Technology Library ISBN: 978-90-386-4029-7

# Acknowledgements

I thank dr. Julien Schmaltz for advising me during the last four years. I have enjoyed our discussions, his feedback, and his cooperation with the papers we wrote together. I am also thankful for the freedom he gave me to pursue various different lines of research. Julien has been very enthusiastic and result driven, which are qualities I value in him as a supervisor.

I thank Prof. Marko van Eekelen. He has supported me on organisational matters. Even when I left the Open University during my PhD in order to follow Julien in joining the TU/e, he continued to have my back. I really appreciate his continued support.

I also thank the other members in my committee, prof. dr. A. Biere, prof. dr. K. G. W. Goossens, prof. dr. T. F. Melham, prof. dr. J. C. van de Pol and prof. dr. ir. J. F. Groote, for reading my thesis and your excellent comments. Armin Biere's comments have greatly helped improve the first three chapters. His results on the complexity classes of several hardware descriptions have also been of great help. Kees Goossens' comments have helped make fundamental improvements to the introduction, which I believe has improved the overall readability of this thesis. I thank Tom Melham for all his comments, especially his suggestion of adding an appendix to this thesis. I thank Jaco van de Pol for pointing out many typos, and for clarifying where the flow of the text needed to be improved. I also thank Jan Friso Groote for his elaborate feedback.

Many thanks to all the supporting staff, especially Chrisja Muris at the Open University, Irma Haerkens and Simone Meeuwsen at the Radboud University, and Margje Mommers and Tineke Bosch at the TU/e. You have helped me find my way through the essential but cumbersome administrative tasks, such that my focus could always be on research.

Freek Verbeek, Bernard van Gastel, Sanne Wouda and Tessa Belder, thank you for working with me, and for our insightful discussions about networks on chips. Special thanks also go out to Jaap van der Woude, and to Bas and Bram Westerbaan, for our elaborate discussions on the more mathematical subjects. Thanks to Freek Wiedijk and Josef Urban for keeping me informed on other theorem provers than ACL2. I thank Bart Jacobs for accepting me in his Digital Security research group, and providing a nice environment for doing research. Baris, Fabian and Rody are thanked for being part of that environment and also a friend. All the

other colleagues at the Radboud University, the TU/e and the Open University are also much appreciated.

I thank Angélique for supporting me through most of my PhD process, that she used to be someone I could come home to. Many thanks to my life-long friend Stijn, for loyally maintaining our friendship while I was busy doing research. Thanks also to my 'paranimfen', Maarten and Rienco, and to Jurjen, Koen, Vic and all my other friends, for their support.

Last but not least, I thank my family. My brother and confidant, Bram, who did an excellent job designing the cover of this thesis. He took care of many details I could not have thought of in a highly professional way. As my thesis had to go to press while I was in Innsbruck, Bram was my feet on the ground in the Netherlands. We had inspiring conversations in the process, and I think the thesis came out looking really nice. My mother, Janny, for reminding me to finish on time, for her love and her emotional support. My father, Stef, for being my coach in the world of academia, a coauthor and fellow researcher in my Ampersand related work, a colleague at the Open University, for being a friend, and, above all, for being my dad.

# Contents

# Chapter 1
# Introduction

The design of microchips is becoming increasingly complex. The size and cost per transistor are going down, and the number of transistors goes up as a result. In 1971, the Intel 4004 had around 2,300 transistors. Fourteen years later (1985), the 386 had 275,000. In 1999, the Pentium 2 Mobile Dixon had over 27 million transistors. The 10-core Xeon (2011) has 2,6 billion transistors. This number is expected to go up with the same pace.

Microchips with such complexity can only be designed from components, called Intellectual Properties (IPs), or IP blocks. Most designs these days are formed through the integration of other designs: we are in an integration era. The functionality we expect from a chip is also growing. The Intel 4004 had an ALU and could fetch instructions from ROM. Later generations added caches and a small code queue. Pentiums have multiple caches and multiple pipelines. Smartphones now have IPs for multiple Central Processor Units (CPUs), graphical processors, GPS units and video decoders. Modern chips feature multiple processors, sometimes even different kinds of processors.

For these components to work together, they need to communicate. With a large number of components, it is not possible or desirable to let all components operate on the same clock frequency. This makes communication between components a complex task, so a component is created for that task. The component that facilitates communication between components is called the interconnect. Often the interconnect is spread throughout the chip, in order for all components to be able to communicate with each other. Such spread-out interconnects are called *communication fabrics*, or *Network on Chips (NoCs)*. An interconnect together with the components is called a System on Chip (SoC).

When designing a SoC, the design will at some point be described at the gate level. The gate level design describes components in terms of their most primitive blocks: gates. Typical gates are the `AND`, `NOT`, `XOR` and `MUX` gate. The `AND`, `NOT` and `XOR` gates correspond to their logical counterparts. `MUX` corresponds to an if-then-else statement. There are also gates which do not necessarily produce a value on all of their outputs, like the `DEMUX`. The `DEMUX` produces its first input at an output that is selected by its second input, but no value – meaning `Z`, or high-impedance – on the other outputs. A gate level design forms a good model for hardware, since it is a description that is present in every hardware design flow.

Ensuring correctness of hardware designs is important. After a system is designed, there are many steps that need to be taken in order to create a finished product. For instance, a gate level description of the design has to be translated into a floor plan. Such a floor plan translates into physical mask structures. These masks then determine where various structures are deposited onto the final silicon. If a design error makes it into silicon, the costs of repairing it will be severe; it is like remaking a stone carving. For instance, Intel had to write off around $1 billion in response to a flaw with Sandy Bridge's SATA companion chips [IntelPR, 2011]. To prevent such issues from occurring, a lot of effort is put into the verification of designs. Nowadays, hardware design teams have around three verification engineers per design engineer.

A trend in hardware verification is to use formal techniques [Kropf, 2013]. Formal techniques make statements about all possible runs of a system, by considering all of its reachable states. Model checking is an example of such a formal technique. The advantage compared to testing is that there can be no untested corner case remaining. This gives the strongest possible guarantee about the assertion. The state of the art in hardware verification is to verify correctness of a component without any testing, also called 'formal sign-off' [Kim et al., 2014]. This means that the confidence in an IP block can be based on formal techniques only, and designs are made final based on this. Existing metrics, such as code coverage, are being adapted for formal [Aggarwal et al., 2011]. A downside to formal verification is that it only works for relatively small components. Scalability remains an issue.

What it means for a hardware design to be correct can be expressed through assertions. Assertions are properties that are written together with the rest of the design. Using assertions to verify a design is called assertion based verification (ABV). A major benefit of ABV is that it allows assertions to be translated through the design process. This way, different tools can check the same assertions in different ways: a computer simulator may run the design together with the assertions and report when it does not hold. When a chip is emulated in a Field Programmable Gate Array (FPGA), the assertions can be checked simultaneously [Boule and Zilic, 2005]. It may even be possible to use the assertions for post-silicon testing [Gao and Cheng, 2010]. On the gate level design, model checkers may symbolically check all possible outcomes of an assertion [Gupta, 1993].

Another method to prove hardware correct, is through the use of a golden model. A golden model is a description of hardware which is known to be correct, or better suited for verification than the gate level implementation. To ensure that a golden model, which may also be a gate level model, corresponds to the gate level implementation, equivalence checkers are used. Checking the correspondence between a golden model and the regular gate level implementation can potentially be very fast. Of course, this leaves open the problem of verifying the golden model itself.

A problem in verification lies with properties that are emergent. These are properties that cannot be stated about a single component in isolation, but depend on the interaction between components. As a result, verification of emergent properties cannot be applied to small isolated components.

'Liveness' is a class of properties that are emergent, and common in intercon-

nects. In general, liveness means that an event will eventually trigger a response. In the context of NoCs, this property would state that when a message is available to a certain component, it will eventually be transferred there. In other words: a message will not get stuck at a certain point. This property becomes interesting when a message may have to wait a while before the next component can accept it. This means that the analysis of the accepting component needs to be included. That component may, in turn, also contain a packet that is waiting to be accepted, and so on. In short, whether or not a network is live may depend on the behavior of all components. There is currently no solution to verify liveness for hardware at the scale of NoCs or communication fabrics. Existing techniques to prove liveness are limited to models of designs, that do not necessarily correspond to the implemented hardware design. When model checking techniques are applied to these liveness properties in gate level designs, they tend to run out of memory, or take much too long to complete. Additional information is required to verify liveness of even relatively small designs [Ray and Brayton, 2012; Ray, 2013].

This thesis gives almost automatic methods to prove liveness. Our contribution uses that hardware is built in components. The general idea is to take a common component, and look at properties about its interface, while hiding its implementation. This means that a hardware designer is required to annotate a component. Using this annotation, the verification can then be automated. We call this 'interface based verification'. In interconnects, a very common component is the queue. For most networks, nearly the entire state of the network can be described through the state of the queues. Annotating the queues allows us to abstract away non-essential implementation details, while still giving us enough information to prove important properties such as liveness. We hide both timing and some data information, so our interface based verification can be seen as a form of behavioral abstraction [Melham, 1990].

This thesis offers solutions to verification engineers, as will be explained in Section 1.3. Based on the annotations, we can derive invariants about the design. Such invariants may improve the scalability of current model checkers. They also enable us to verify liveness of a design. These are analyses previously performed with the help of a high-level model, which can now be performed through the use of queue annotations only.

Before discussing the outline of this thesis, we give some background into several hardware descriptions, and existing verification techniques.

## 1.1 Hardware descriptions

A hardware design language is a language in which to model hardware. Two major hardware design languages are Verilog and VHDL [IEEE, 2001, 2009]. The examples in this thesis will be in Verilog. Verilog has some features that make it convenient as a programming language. For example, a `generate` statement can be used like a `for`-loop in C or Java, which will create several similar structures, one for each iteration of the loop. As a consequence of these features, it is possible to create very small Verilog code fragments to describe very large structures in hardware. It is even possible to use statements that do not correspond to any hardware at all, but may be useful when simulating the hardware on a computer.

Below is an example of some Verilog code. It is just meant to give an impression about the kind of statements that are possible in Verilog. This code writes, at each clock tick, to address `i`, for `LENGTH` different values of `i`, starting from 0. If the value of `in` is equal to `i`, and `writing` is set to high (i.e. true), the value written is `i0.data`. If not, the old value is written, such that the value remains unchanged:

```
generate
  genvar i;
  for (i = 0; i < LENGTH; i = i + 1)
     always @(posedge clk)
        data[i] <= ((writing && in==i)
                 ? i0.data : data[i]);
endgenerate
```

Aside from describing hardware in a language which humans can write, we also look at hardware in descriptions which can be analysed more easily. The previous section mentioned gate level hardware as such a description. Gate level hardware can be seen as using a very restricted subset of Verilog. In particular, gate level Verilog only allows wire declarations, and Verilog statements in which either a module or a gate is instantiated. For the example given above, if `LENGTH` is 2, the gate level Verilog could be the following:

```
register r1 (.d(n40), .clk(clk), .s(1'b0), .r(1'b0),
  .q(\data[0] [0]));
not (n38, in);
and (n39, writing, n38);
mux (n40, n39, i0_data[0], \data[0] [0]);
register r2 (.d(n45), .clk(clk), .s(1'b0), .r(1'b0),
  .q(\data[1] [0]));
and (n44, writing, in);
mux (n45, n44, i0_data[0], \data[1] [0]);
```

Intermediate expressions have been translated to additional wires. Register `r1` is a module instance of a single register, driven by `n40`. The wire `n40` represents the expression `n39 ? i0_data[0] : \data[0] [0]`, since it is the output of the `mux` gate on the fourth line. Similarly, `n39` stands for `writing && n38`, where `n38` stands for the negation of `in`. Turning intermediate expressions into gates makes the code harder to read, but easier to parse. By using gate level Verilog as input for our tools, only a limited set of primitive gates needs to be supported.

The take-away message from this example, is that instantiation statements in gate level Verilog fall into two categories: (1) statements in which a built-in gate is instantiated, such as the `and` gate, or the `not` gate, and (2) statements in which another module is instantiated, such as the `register` statements. By allowing modules to be instantiated, it becomes explicit when the same hardware design occurs in multiple places. For instance, this makes it possible for us to easily identify all queues in a design.

Another advantage of this language, is that it can be generated automatically from the full Verilog language. In this translation, blocks like `generate` will expand into several repetitions of the inner statements. A statement that does

not have a gate level counterpart, will throw a warning or an error. Code that can be translated into gate level descriptions are called synthesizable. The name 'synthesizable' stems the fact that the translation from Verilog to a gate level description is called 'synthesis'. We tend to avoid this name, as most synthesis tools also perform optimizations, such as reducing the number of gates produced, and do not necessarily produce Verilog as output. In this thesis, we use 'Verific'[1] to translate Verilog to gate level Verilog, for its ability to parse industrial Verilog, its industrial use [Haynal et al., 2008], and its free academic license.

Gate level descriptions are not the only descriptions used for verification. As input for model checkers, it is important that the 'next state' of a system is specified. For hardware, the next state can be described through a function that gives the next value for each register. This description is called Register Transfer Level (RTL).

RTL is a useful format for verification, and a convenient way to describe hardware. The following Verilog code would correspond with an RTL description:

```
always @(posedge clk) begin:
 data[0] <= (writing && in==0) ? i0.data : data[0];
 data[1] <= (writing && in==1) ? i0.data : data[1];
end
```

We found a class of gate level Verilog programs that could not be translated into RTL by existing tools. While engineers may be able to work around this in many cases, we found that busses fall into this class that could not be translated. Since busses are common structures in interconnects, we decided to remedy this problem. We propose our own translation from gate level to RTL in Chapter 3.

## 1.2 Hardware verification

This section looks at formal hardware verification techniques.

In the introduction, we saw that certain properties of hardware could be checked, using model checking techniques. *Model checkers* contain algorithms that can decide for a large class of properties whether or not they hold. These approaches have gained a lot of popularity, as they do not require additional user input. Model checkers determine whether some model of a system satisfies a certain specification. The model is described as a state machine, so hardware is usually represented at the RTL. The specification is described in a temporal logic. A model checking algorithm searches a state, or sequence of states, that does not satisfy the specification [Biere et al., 2003]. When found, the user can be presented with a counterexample. If such a counterexample is not found, the user is certain that the specification holds.

Model checkers sometimes do not give an answer for certain problems, even if these fall into the language required for them. The issue is that a model checker may run out of memory, or will take so long that its execution gets aborted. There are several ways to mitigate this issue.

[1] http://www.verific.com/

*Theorem proving* refers to the technique of writing down a mathematical proof in such a detailed way, that a computer program can check it. The required user input is a full proof. These proofs are established in close collaboration with the computer program that checks it, in a process called interactive theorem proving. A major benefit of theorem proving is that it allows a very large class of properties to be proven. It is applied to gate level hardware designs [Hunt Jr, 1989; Hunt and Swords, 2009]. Unfortunately, interactive theorem proving is seen as a technique which requires a lot of manual effort, and highly skilled users.

Another way to find an answer, is by *adding an assumption* to the model. Some model check algorithms use a Satisfiability (SAT) encoding as an over-approximation of the reachable state space. If an assumption is added to the model, this assumption strengthens that SAT encoding. This strengthening can help the model check algorithm derive that no counterexample can be found. Model check algorithms that use 'explicit state model checking' only keep an under-approximation of the reachable state space. The additional assumptions are less likely to help with these algorithms, although in rare cases the encoding of the under-approximation might be somewhat simplified. When it is possible to prove the desired property with the added assumption, the assumption becomes an assertion that needs to be verified as well. This adds a new assertion to the list of assertions that needs to be verified. It is therefore important to choose the assumptions such that they hold, and are easy to verify. Adding assumptions can be a trial and error process, and is used a lot in practice.

*Equivalence checking* refers to a technique of comparing two hardware models that are described at the same level. This is useful when a different description of the hardware is easier to verify. It can also be used when such different description has already been verified, but changes have been made to it later on. In equivalence checking, the model which is known to be correct is called the golden model, in order to distinguish it from the regular model.

A common technique that is incorporated in equivalence checking is SAT sweeping[Kuehlmann, 2004; Zhu et al., 2006]. This method uses information about which registers in one model correspond to which registers in the other. Two models correspond if the inputs of the corresponding registers will always hold the same value in both models, given that the external inputs and the outputs of the registers are the same. SAT sweeping iterates over all wires and decides which wires correspond. Often there is a structural correspondence. For example, say wire $w$ is the output of an `and` gate of wires $a$ and $b$ in one design, and $a$ and $b$ correspond to the wires $a'$ and $b'$ in the other. In that case, the output wire of the `and` gate of wires $a'$ and $b'$ is known to correspond to $w$. If there is no structural correspondence, a set of test cases is run to check whether correspondence can be ruled out. If not, a call to a SAT solver is made. The question whether two wires can hold a different value is posed to the SAT solver. If not, the wires correspond. If so, the SAT solver will respond with a new test, which is able to distinguish between the two wires. When the two hardware models are similar, especially when the registers in the two models have a one-to-one correspondence, equivalence checking is very fast, making it a common method in the hardware design industry.

*Validation* can also be seen as a verification technique. Definitions of what validation means vary. Here, we position it as deciding if two descriptions made

on different levels of abstraction correspond. A typical way to verify this, is to automatically translate one level to the other, and verify their equivalence. Typically the more abstract level model would be translated into the more concrete one, called generation. If the translation is simple or independently verified, the generated model would be 'correct by construction'. In this case, the generated model plays the role of a golden model in equivalence checking. Another way to verify the correspondence, is to generate assertions from more abstract model, and verify them on the more concrete one. It may be the case that the more abstract model only exists in terms of such assertions, making this validation strategy a very obvious one. Validation is a type of verification that often involves many manual steps, but it is such a crucial one that it is commonly seen in the hardware design industry.

Figure 1.1 shows some of the types of verification that are typically seen. A high level idea is created by a hardware architect, which is communicated to design engineers in order for it to be implemented. In the mean time, verification engineers write assertions corresponding with the general ideas of the architect. Once an initial implementation has been produced, it can be checked against the assertions using model checkers. If an assertion does not hold, either the implementation or the assertion must be changed. In some cases, this can be because the high level model was wrong. If the model checker cannot verify the assertion, other approaches are taken. For instance, a verification engineer can add assumptions to the model, or work with designers to create a golden model that can be verified. The work in this thesis can help verification engineers in these approaches, and provides an alternative way to verify liveness. We will go into details on the 'contribution' part of this figure in Section 1.3.

## Liveness

As discussed previously, liveness means that an event will eventually trigger a response. Previous attempts to verify liveness include attempts to improve the performance of model checkers using information that is not available directly from the Verilog description. They only work on relatively small examples. Liveness has been verified on RTL descriptions of networks, provided that the additional information is available. This is best illustrated in the PhD thesis of Sayak Ray [Ray, 2013], and in research by Alexander Gotmanov et al. [Gotmanov et al., 2011]. One of the key properties used by both researchers, is that invariants need to be derived automatically from a separate description of NoCs, in a language called xMAS. Aside from the benefits that these invariants can be generated from xMAS, it also allows RTL implementations to be generated from it. This is shown to speed-up hardware model-checking [Chatterjee and Kishinevsky, 2012; Ray and Brayton, 2012]. The xMAS language will be explained in Chapter 2. We will discuss these invariants in Chapter 4.

In the case of interconnects, liveness cannot be checked in isolation. This is shown in work by Freek Verbeek and Julien Schmaltz [Verbeek and Schmaltz, 2012a,b]. In this work routing topology is analysed separately from the switching scheme in a formal analysis of NoCs at an abstract level, called GeNoC. Many properties can be verified in isolation, but for liveness verification the topology and

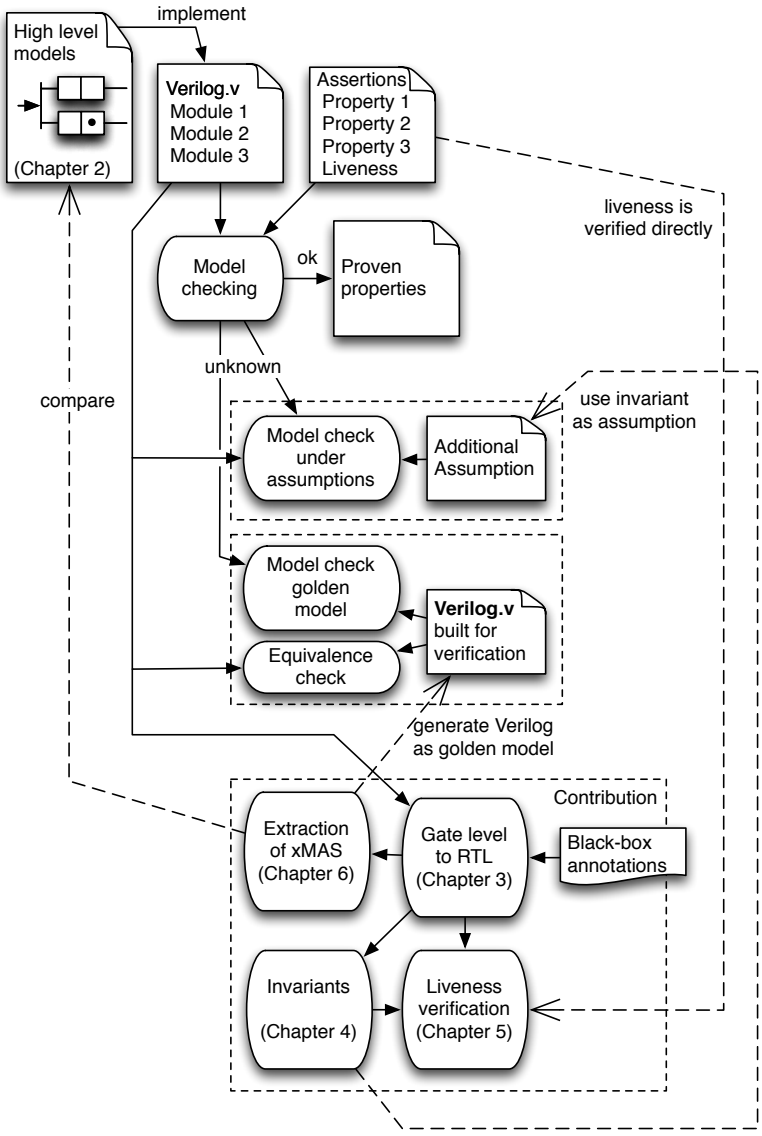



**Figure 1.1:** *A summary of how the tool support is embedded in the design flow. Arrows represent which information is used for which algorithms. Dotted arrows represent new verification possibilities.*

switching scheme need to be combined. The authors recognize that the switching scheme, network topology and routing all contribute to the behavior of a NoC. A later review by Balaji Venu and Ashwani Singh [Venu and Singh, 2012] confirms this, and identifies liveness as the main challenge in the design of a NoC.

Some work on liveness verification focusses on a specific interconnect architecture. The Ætheral protocol as described by Kees Goossens [Goossens et al., 2005], for instance, has been proven live (free of deadlocks in their terminology) with the use of the theorem prover PVS in [Gebremichael et al., 2005]. The argument for liveness is similar to that of José Duato, for his NoC design language [Duato, 1993]: given the dependencies, if there is no circular wait, there cannot be a deadlock [Duato, 1995]. It was recently noted that although the argument is presented as necessary and sufficient, it is not necessary, meaning that there are deadlock-free circuits in Duato's language that do have circular waits [Verbeek and Schmaltz, 2011a].

## 1.3 Contribution of this thesis

In this thesis, we aim to verify interconnects, with the analytical strength and performance of algorithms that operate at an abstract level, on a hardware design described at gate level. Certain components, such as queues and routers, are incredibly common in communication fabrics. They are so common that in many NoCs, they are in fact the only state-holding elements. Since routers do not 'hold' packets, the notion of a 'packet in a network' can be expressed through the notion of a packet in a queue. It can be difficult to define what it means for a packet to be in a network, expressed in terms of RTL code. At a high level, it is often clear: a queue has a certain number of places, and for each place that contains a packet, that packet is in the queue.

A convenient form of verification is done at the RTL, provided that we keep relevant module information. As mentioned, at the gate level we allow for hardware to be expressed in terms of modules. Queues are often expressed as separate modules. This makes it possible for us to consider the queue at a high level, while considering the rest of the RTL at a low level. To do so, we hide the RTL implementation of a queue.

A module, of which the implementation is hidden is called a black box. The remaining hardware still has an interface with the black box, over wires. Conceptually, this interface is very similar to the interface the NoC has with its environment: the NoC has a set of input wires from which it gets information, and a set of output wires to which it can write information. The environment may then react to that information. Note that the input wires of the NoC are output wires of the environment, and vice versa. Similarly, the output wires of a black box are considered to be input wires of the NoC, and vice versa.

The main difference between in- and output wires with the environment, and those with a black box, is the scope of the wire names. This is illustrated with an example in Figure 1.2. The black box, called `queue`, is used once in the module `intermediate`. Here it has the instance name `wait`. The module `intermediate` is used twice in the top level module called `top`. Each use of the module `intermediate` has a different name: `inst1` and `inst2` respectively. To

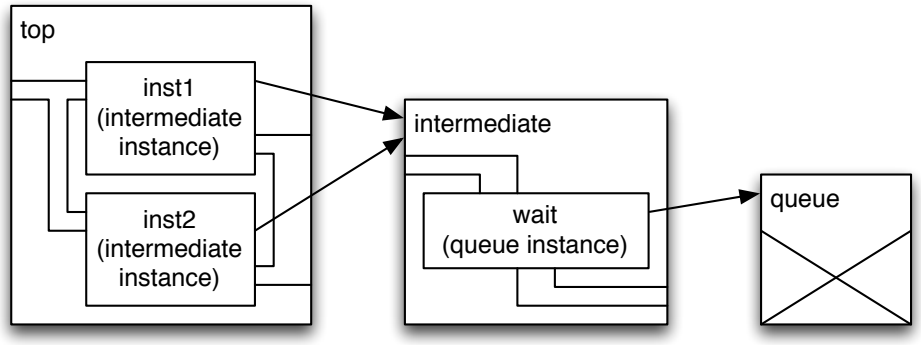



**Figure 1.2:** *The black box 'queue' is used once in all instances of 'intermediate'. As 'intermediate' occurs twice in 'top', the black box module also occurs twice in 'top'.*

distinguish the two occurrences of `queue`, we take into account the entire 'path' from `top` to `queue`, through the instance-names. For instance, `inst1/wait` and `inst2/wait` would be the path for the queues in our example. These paths uniquely identify every queue instance. If the queue module has the output wire `not_full`, we can treat `inst2/wait/not_full` as a free variable. Similarly, if the module has the input wire `enqueue`, the module described in the figure assigns a value to `inst2/wait/not_full`, giving it the role of an output value relative to the top level module.

To translate gate level designs to RTL designs, the usual approach is to forbid designs for which a straight-forward translation is not possible. In the previous example, we need to know whether `not_full` is an input or an output in order to decide whether or not to treat it as a free variable. Problems arise when a wire can be both an input and an output. Unfortunately, these situations do arise in practice. In Chapter 3, we show how such gate level designs can be translated to RTL designs.

When proving properties about designs with a large state-space, invariants help constrain this state-space. Invariants are typically seen as a property that is easy to obtain from abstract descriptions of hardware, but very difficult to obtain from RTL designs. They play a crucial role in proving emergent properties, such as liveness. Chapter 4 describes a novel method to obtain invariants at RTL automatically, which aids the model checking process. This chapter is based on a paper published at MEMOCODE in 2013 [Joosten and Schmaltz, 2013]:

> Joosten, S. J. C. and Schmaltz, J. (2013). Generation of inductive invariants from register transfer level designs of communication fabrics. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 57–64. IEEE

As discussed in Section 1.2, liveness is an important property to check about interconnects. Chapter 5 presents a novel technique to prove liveness through the

absence of local deadlocks on RTL designs automatically. This chapter is based on a paper published at DATE in 2014 [Joosten and Schmaltz, 2014]:

> Joosten, S. J. C. and Schmaltz, J. (2014). Scalable liveness verification for communication fabrics. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 113:1–113:6

A general challenge is to use our interface based technique to verify everything that can potentially be verified at an abstract level. Chapter 6 is a first attempt to address that challenge. It shows how to derive an xMAS network from the RTL netlist. This research was published at DATE in 2015 [Joosten and Schmaltz, 2015]:

> Joosten, S. J. C. and Schmaltz, J. (2015). Automatic extraction of micro-architectural models of communication fabrics from register transfer level designs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 1413–1418

For this thesis, several algorithms for the analysis of circuits are implemented. The 'Contribution' box in Figure 1.1 gives an overview of how these algorithms relate. The implementations are tested in Chapter 7. This chapter combines the experimental results section of several published papers [Joosten and Schmaltz, 2013, 2014, 2015], together with some new experimental results.

Chapter 8 combines the future work section of several papers, and gives a more thorough outlook based on the most recent experiences.

Appendix A describes the application of functional programming techniques to create a framework for the analysis of Verilog designs. The chapter is intended as a reference for anyone interested in using, improving or rebuilding the framework. The described implementations are by the author of this thesis, and can be downloaded from the link given in the chapter.

The methods in this thesis look at interconnects at the gate level and the RTL. To be able to easily convey the design of interconnects, we describe them in an abstract model. To describe interconnects, we use the language xMAS, for several reasons. First, it has a straightforward gate level implementation. Second, it allows us to compare our methods to those that require the xMAS models.

The xMAS language and notation is introduced in Chapter 2. This chapter is largely based on work by Chatterjee et al. [Chatterjee et al., 2012, 2010]. We also use this chapter to introduce the queue annotations that are used throughout this thesis.

## Work that is not presented in this thesis

The author of this thesis contributed to a paper about automatic deadlock verification for asynchronous click circuits. The xMAS language we describe in Chapter 2 can be seen as a synchronous version of click circuits. For that reason, liveness

verification methods that work on xMAS turn out to be possible on click circuits as well. This paper deviates slightly from the line presented in this thesis, and contains a significant body of work that was solely done by Freek Verbeek. For these reasons, the following paper is omitted from this thesis [Verbeek et al., 2013]:

> Verbeek, F., Joosten, S. J. C., and Schmaltz, J. (2013). Formal deadlock verification for click circuits. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 183–190. IEEE

The author of this thesis helped to give process algebra semantics to xMAS components. The main contribution of the corresponding paper, is that it shows how to use model checking at the xMAS level. This work with Sanne Wouda is omitted from this thesis entirely, but published separately [Wouda et al., 2015]:

> Wouda, S., Joosten, S. J. C., and Schmaltz, J. (2015). Process algebra semantics & reachability analysis for micro-architectural models of communication fabrics. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 Thirteenth ACM/IEEE International Conference on*

A tool for designing interconnects using xMAS is presented in [Joosten et al., 2014b]. This tool, called 'Wicked xMAS' was implemented by several students (some of which I supervised) over the course of several years, and is maintained by Freek Verbeek, Bernard van Gastel, Julien Schmaltz and the author of this thesis. Use of Wicked xMAS has helped reproduce some xMAS designs, some of which are used throughout this thesis. This thesis contains some details about xMAS, but we could not find a satisfactory place to describe the tool Wicked xMAS, which is why the paper is omitted from this thesis:

> Joosten, S. J. C., Verbeek, F., and Schmaltz, J. (2014b). WickedXmas: Designing and verifying on-chip communication fabrics. In *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*

The papers [Joosten and Zantema, 2013; Joosten et al., 2014a, 2013; Joosten and Joosten, 2015] are based on research by the author, but were omitted from this thesis because they have no obvious relation to communication fabrics:

> Joosten, S. J. C. and Zantema, H. (2013). Relaxation of 3-partition instances. In *CTW*, pages 133–136

Joosten, S. J. C., Kaliszyk, C., and Urban, J. (2014a). Initial experiments with TPTP-style automated theorem provers on ACL2 problems. In *International Workshop on the ACL2 Theorem Prover and its Applications*, volume 152, pages 77–85

Joosten, S. J. C., Van Gastel, B., and Schmaltz, J. (2013). A macro for reusing abstract functions and theorems. In Gamboa, R. and Davis, J., editors, *International Workshop on the ACL2 Theorem Prover and its Applications*, volume EPTCS 114, pages 29–41

Joosten, S. and Joosten, S. J. C. (2015). Type checking by domain analysis in ampersand. In *RAMICS 2015, 15th International Conference on Relational and Algebraic Methods in Computer Science, Braga*

# Chapter 2
# Micro-architectural models of communication fabrics

To formally verify communication fabrics, *micro-architectural* models are commonly used. A graphical language, xMAS, was recently proposed by Intel [Chatterjee et al., 2012]. It is intended to specify such micro-architectures formally. For this language several properties can be proven automatically for reasonably large networks, such as invariants [Chatterjee and Kishinevsky, 2012], channel type information [Van Gastel et al., 2014] and deadlock freedom [Verbeek and Schmaltz, 2011b].

Another way to use high-level models, is to improve the effectiveness of hardware model checking [Chatterjee and Kishinevsky, 2012; Gotmanov et al., 2011; Ray and Brayton, 2012]. Hardware designs analyzed in the aforementioned works are generated from xMAS models, such that the correspondence between the hardware which is checked, and the xMAS models used is clear. In practice, high-level models are difficult to create and their relation to actual designs can be unclear.

We introduce the reader to the part of the xMAS specification relevant to this work.

## 2.1 Definition of xMAS

An xMAS model is a network of primitives connected via typed *channels*. A channel is connected to an *initiator* and a *target* primitive. To indicate whether a primitive acts as an initiator or a target, we say a channel is an *output* channel of the initator, or an *input* channel of the target. A channel is composed of three signals. Channel signal $c$.irdy indicates whether the initiator is ready to write to channel $c$. Channel signal $c$.trdy indicates whether the target is ready to read from channel $c$. Channel signal $c$.data contains data that are transferred from the initiator output to the target input. Data is transferred if and only if both signals $c$.irdy and $c$.trdy are set to true.

Figure 2.1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert packet types and represent message dependencies inside the fabric or in the model of the

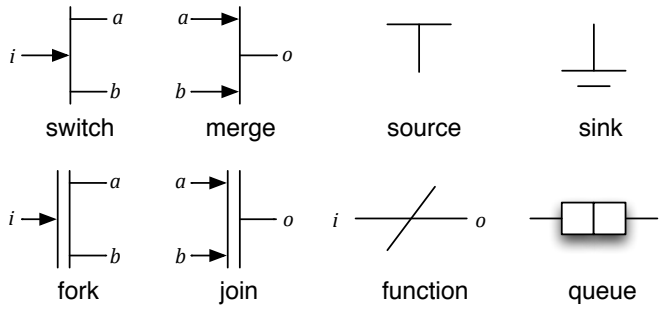



**Figure 2.1:** *There are eight xMAS components.*

environment. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. The arbitration policy is a parameter of the merge. A *queue* stores data. Messages are non-deterministically produced and consumed at *sources* and *sinks*. Below are the original definitions for fork, join, switch and merge [Chatterjee et al., 2012]:

**fork** Duplicates a packet. It has output channels $a$ and $b$, and input channel $i$:

$$\begin{aligned} a.\text{irdy} &:= i.\text{irdy} \wedge b.\text{trdy} \\ b.\text{irdy} &:= i.\text{irdy} \wedge a.\text{trdy} \\ i.\text{trdy} &:= a.\text{trdy} \wedge b.\text{trdy} \\ a.\text{data} &:= i.\text{data} \\ b.\text{data} &:= i.\text{data} \end{aligned}$$

**join** Combines two packets. For input channels $a$ and $b$, and output channel $o$:

$$\begin{aligned} a.\text{trdy} &:= o.\text{trdy} \wedge b.\text{irdy} \\ b.\text{trdy} &:= o.\text{trdy} \wedge a.\text{irdy} \\ o.\text{irdy} &:= a.\text{irdy} \wedge b.\text{irdy} \end{aligned}$$

The $o$.data value depends on an additional function, which takes $a$.data and $b$.data as arguments.

**switch** Routes a packet to one of its output channels. For output channels $a$ and

$b$, and input channel $i$:

$$\begin{aligned} a.\text{irdy} &:= i.\text{irdy} \wedge s \\ b.\text{irdy} &:= i.\text{irdy} \wedge \neg s \\ i.\text{trdy} &:= (a.\text{trdy} \wedge s) \vee (b.\text{trdy} \wedge \neg s) \\ a.\text{data} &:= i.\text{data} \\ b.\text{data} &:= i.\text{data} \\ s &:= f(i.\text{data}) \end{aligned}$$

Note that $s$ is used as a switching function which should only depend on the input data.

**merge** Selects a packet from one of its input channels. For input channel channels $a$ and $b$, and output channel $o$:

$$\begin{aligned} a.\text{trdy} &:= o.\text{trdy} \wedge u \wedge a.\text{irdy} \\ b.\text{trdy} &:= o.\text{trdy} \wedge \neg u \wedge b.\text{irdy} \\ o.\text{irdy} &:= a.\text{irdy} \vee b.\text{irdy} \\ o.\text{data} &:= \begin{cases} a.\text{data} & \text{when } u \\ b.\text{data} & \text{otherwise} \end{cases} \end{aligned}$$

The original definition states that $u$ is 'a local state variable that ensures fairness', where fairness means that if $o$.trdy is high infinitely often, $a$ and $b$ get infinitely many turns: $a$.trdy and $b$.trdy will be high eventually. This condition prevents starvation.

**function** Changes the data:

$$\begin{aligned} i.\text{trdy} &:= o.\text{trdy} \\ o.\text{irdy} &:= i.\text{irdy} \end{aligned}$$

The $o$.data value is given by an additional function, which takes $i$.data as an argument.

**sink, source** These generally represent components which are abstracted away from, or external in- and outputs to the network. As such, the source has $i$.irdy and $i$.data as free input variables. The sink has $i$.trdy as free input variables. For the source, when .irdy is high and .trdy is not, the .data signal does not change at the next clock tick.

Sometimes a sink or a source is always ready to send a packet. This is indicated by adding the word 'eager' near the sink or source.

**eager source** Is always ready to send a packet.

$$o.\text{irdy} := \mathit{true}$$

**eager sink** Is always ready to receive a packet.

$$i.\text{trdy} := \mathit{true}$$

**queue** We will describe the queue in greater detail in the next section.

### 2.1.1 Queue

We give an implementation of a four-place circular queue in this section. Using this queue, and similar ones, we construct the networks on which we focus our examples and experiments throughout this thesis. We illustrate the interaction between these components in Section 2.1.2, by giving a small example.

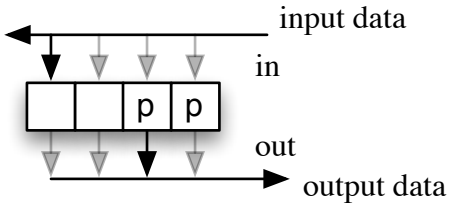



**Figure 2.2:** *A four place circular buffer. This figure shows a situation with two packets, indicated by a 'p'.*

The implemented queue is a circular queue (Figure 2.2). An advantage of using a circular queue is that it reduces the writes to registers inside it: every time a register changes its value, some amount of energy is consumed. A queue that minimises writes to registers, also reduces power consumption.

The idea behind a circular queue is to use two pointers: one called `in` to indicate a slot where the next packet should be written, and one called `out` to indicate where the current output data should come from. The queue presented here has four places, so we use a two bit register `in` to keep track of where the packet entering next should be placed. Another two bit register `out` keeps track of which packet is offered at the output side. In Verilog, the statement `reg [1:0] in;` declares `in` as a two bit register.

When the read data, called `o$data`, is set, the queue waits until it is read. To indicate this, `o$irdy` is set to high. This is acknowledged by the environment by a high `o$trdy` value. For `o$data` and `o$trdy`, the `$` sign is used in the place of a dot, as a dot is not allowed in unescaped Verilog identifiers. The environment is reading when `o$irdy` and `o$trdy` are both high. In Verilog, we use the local variable `reading` to indicate this: `wire reading = o$irdy && o$trdy;`.

In our queue implementation, instead of erasing the data, the `out` pointer is increased with 1. Updates which should be made on the next clock tick are indicated by a <= in Verilog, and should always be inside an `always` block, to indicate which clock tick. Within such blocks, we can take care of resets as well.

We use a two-bit local variable `nextout` to indicate the next value of the `out` pointer in case it is read. A local wire can be declared and assigned in one Verilog statement: `wire [1:0] nextout = ...;`. An update of the `out` pointer is written as: `out <= rst ? 0 : (reading ? nextout : out);`.

If the queue is full or empty, `in` and `out` point to the same position. The distinction between full and empty queues is made by an additional single-bit register: `reg full;`.

Every time a packet is injected (input is ready and queue is not full), `in` is

raised by one. When a packet is taken (output is ready and queue is not empty), `out` is raised by one. When the two are equal, the queue is either full or empty. To distinguish between the two cases, a one-bit-register `full` is set to high if `in` is raised to be the value of `out` while no packet is taken. On the input, .trdy is defined as the negation of `full`. In Verilog: `assign i$trdy = !full;`. If output-ready is low, no packet enters the queue, even if a packet is taken from the turn of that cycle (so latency increases if the full capacity of the queue is used).

For this implementation, the data is 8 bits. Therefore `i$data` and `o$data` are declared as 8-bit in and output wires by their range `[7:0]`.

```
module circular_queue_explicit (clk, rst, i$irdy, i$trdy,
    i$data, o$irdy, o$trdy, o$data);
input clk, rst, i$irdy;
output i$trdy;
input [7:0] i$data;
output o$irdy;
input o$trdy;
output [7:0] o$data;

reg [1:0] in, out;
reg [7:0] data0, data1, data2, data3;
reg full;

assign i$trdy = !full;
assign o$irdy = !(in==out) || full;
assign o$data = out==0 ? data0 :
              (out==1 ? data1 : (out==2 ? data2 : data3));

wire writing = i$irdy && i$trdy; // writing into the queue
wire reading = o$irdy && o$trdy; // reading from the queue
wire [1:0] nextin = in==3 ? 0 : (in+1);
wire [1:0] nextout = out==3 ? 0 : (out+1);

always @(posedge clk) begin
 in <= rst ? 0 : (writing ? nextin : in);
 out <= rst ? 0 : (reading ? nextout : out);
 full <= (rst || reading) ? 0 : ((nextin==out && writing)
     ? 1 : full);
 data0 <= (writing && (in==0)) ? i$data : data0;
 data1 <= (writing && (in==1)) ? i$data : data1;
 data2 <= (writing && (in==2)) ? i$data : data2;
 data3 <= (writing && (in==3)) ? i$data : data3;
end
endmodule
```

Note that by the definition of `reading` and `writing`, the queues are the components that ensure that a packet is transferred if and only if both .irdy and .trdy signals are high. Also, the queues ensure that the data does not change if

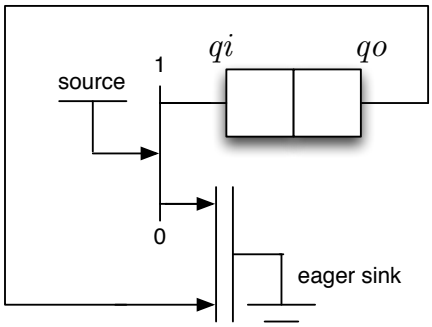



**Figure 2.3:** *A small xMAS network, to help explain how* .irdy *and* .trdy *values propagate.*

.irdy is high, and no transfer occurs.

### 2.1.2 Composition of components

When xMAS components are combined using channels, .irdy and .data values are propagated in the direction of the arrows, while .trdy is propagated in the other direction. Figure 2.3 gives a small example as to what this looks like at the interface of a queue. In this example, the source can inject packets with data 0 or 1 in order to decrease or increase the number of packets in the queue. The sink in this example will always accept packets, thus it is eager.

The queue has three inputs-wires: $qi$.irdy, $qi$.data and $qo$.trdy. These are assigned values by the network. In the example given, the value of $qi$.irdy depends on the value given by the switch, which was defined as: $i.\text{irdy} \wedge s$. The select function $s$ is true if the data on its input is high, and the value $i$.irdy is given by the source: source.irdy. Consequently, the value of the $qi$.irdy wire at the input of the queue is given by:

$$qi.\text{irdy} = \mathsf{source}.\text{irdy} \wedge (\mathsf{source}.\text{data} = 1)$$

The .data signal to $qi$ is rather trivial. While a transfer can only occur at the input of the queue if $qi$.irdy is high, and therefore source.data = 1, the value of $qi$.data can still be 0:

$$qi.\text{data} = \mathsf{source}.\text{data}$$

Similarly, $qo$.trdy depends on the join, switch and sink:

$$qo.\text{trdy} = \mathsf{source}.\text{irdy} \wedge (\mathsf{source}.\text{data} = 0)$$

## 2.2 Non-standard xMAS components

Many nice examples can be made using the standard xMAS components. This work shows types of analysis that apply to non-xMAS networks as well. For this reason, we do not only use xMAS components, but also some components that do not exist within xMAS.

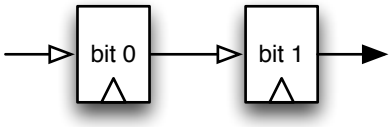



**Figure 2.4:** *Two single registers that behave as one-place queues with no data.*

A simple two-bit shift counter is shown in Figure 2.4. The idea is that when .irdy is high at the input, the packet is accepted by setting the first bit – bit 0 – to high, while data is not stored. That first bit must become low again before another packet can be accepted, which can happen by setting the second bit – bit 1 – to high. That second bit will wait for a high .trdy at the output before becoming low. The Verilog is as follows:

```
module counter (clk, rst, i$irdy, i$trdy, o$irdy, o$trdy);

input clk, rst, i$irdy, o$trdy;
output i$trdy, o$irdy;
reg bit0, bit1;

assign i$trdy = !bit0;
assign o$irdy = bit1;

wire writing = i$irdy && i$trdy;
wire reading = o$irdy && o$trdy;

always @(posedge clk)
  bit0 <= rst ? 0 : (writing ? 1 : (bit0 ? bit1 : 0));
always @(posedge clk)
  bit1 <= rst ? 0 : (reading ? 0 : (bit1 ? 1 : bit0));

endmodule
```

Note that the interface to this module is similar to that of a queue, so this component can be used within xMAS networks.

### 2.2.1 Interfaces in RTL

In the previous section, we saw how we could wrap non-xMAS elements inside an xMAS wrapper. When such wrappers do not exist, we still analyse the hardware. In order to do so, there should be *some* agreement on when a packet enters the queue, and when it leaves. The property which encodes when a packet is put into the queue is called $t_r$. The property that encodes when a packet leaves the queue is called $t_s$. These properties do not need to exist as wires in the physical hardware, but it should always be possible to express them in terms of wires that are in the hardware.

We make clear what we mean by a property through a preliminary definition. In the next section, we generalise this definition slightly to account for black boxes.

We focus on synchronous RTL: all registers can be perceived as being updated simultaneously, by an implicit clock. While we write RTL in Verilog, we illustrate our approach using an abstraction as a starting point. The disjoint union between sets is written as $\oplus$, and the set of Boolean values as $\mathbf{2}$.

**Definition 2.1** (Circuit without black boxes, state, property) A *circuit without black boxes* consists of (a finite set of) inputs $I$ and registers $F$, together with a next-step function $n : (I \oplus F \to \mathbf{2}) \to (F \to \mathbf{2})$. A *state* $\sigma$ of a circuit is a function $\sigma : I \oplus F \to \mathbf{2}$ which assigns a value to every input and every register. A *property* $p$ in a circuit is a function which, given a state, produces a value $p : (I \oplus F \to \mathbf{2}) \to \mathbf{2}$.

By this definition, the value of inputs is part of the state, such that a property only depends on the state of the network. The value at an output wire of a chip can be considered a property.

For a network to function correctly, it should not attempt to write to full queues, or read from empty queues. To prevent the network from doing this, a queue could indicate when it is ready to receive, $r_r$, or to send, $r_s$. These wires, however, need not exist. Suppose, for example, that $r_r$ is not available. In this case, the hardware must ensure that a packet can never be written to the queue if it is full. For the queue, this means that it must be of a size that is sufficient for where it is used. If this can be asserted, we may use *true* as the expression for $r_r$. In other words: it is always possible to give an expression for $r_r$ and $r_s$.

In summary, a queue interface can be identified by the following (Boolean) properties:

$r_r$ The queue is ready to receive.

$r_s$ The queue is ready to send.

$t_r$ A packet is added to the queue at the next tick.

$t_s$ A packet is sent at the next tick.

The queue interface satisfies $t_r$ implies $r_r$ and $t_s$ implies $r_s$, meaning that whenever a packet is enqueued, the queue must be ready for it, and whenever a packet is dequeued, the queue should have been ready. In most RTL designs, this is true. Note that this also holds when $r_r$ or $r_s$ (or both) are simply defined as true. Together, $t_r$ and $r_r$ give rise to a 'port', as do $t_s$ and $r_s$. The former – on the receive side – is the input port of a queue, the latter is the output port. The term *port* is defined as follows:

**Definition 2.2** (port) A *port* (in a circuit) is an object $x$ for which properties $r_x$ and $t_x$ are defined. We require that $t_x(\sigma) \to r_x(\sigma)$ for every state $\sigma$, and refer to $t_x$ as the transfer property, and to $r_x$ as the implied property.

For an xMAS queue, the $r$ and $s$ ports are defined as follows:

$$r_r = i.\text{trdy}$$
$$t_r = i.\text{trdy} \wedge i.\text{irdy}$$
$$r_s = o.\text{irdy}$$
$$t_s = o.\text{trdy} \wedge o.\text{irdy}$$

Note that we can have more information about a queue. For instance, it is possible to have information about the data going in and out of a queue. We write $d_x[i]$ to indicate data-bit $i$ on port $x$.

### 2.2.2 Treating queues as black boxes

In hardware, a module is called a black box when its interface is given, but its implementation is not. Black boxes are often used in the design process of hardware. In this thesis, we also use them for the analysis of hardware. By using a black box, we hide irrelevant details from the analysis.

A downside to using a black box, is that some relevant features may also be missing. In this thesis, each queue will be treated as a black box. Consequently, we need a way to find out the values of $r_r$, $t_r$, $r_s$ and $t_s$ for each queue. This is achieved by annotating queues: stating that some module is to be treated as a black box, while adding the necessary extra information.

For our tool, Voi, we annotate the module `xmas_queue_2`, a queue that can hold 2 packets, and `xmas_queue_4`, of size 4, with the following syntax:

```
annotate queue; // Queue sizes 2 and 4
  module(.upperBound(2)) xmas_queue_2;
  module(.upperBound(4)) xmas_queue_4;
  inputReady i$trdy;
  inputTransfer i$trdy && i$irdy;
  inputData i$data;
  outputReady o$irdy;
  outputTransfer o$trdy && o$irdy;
  outputData i$data;
endannotate
```

The value $r_r$ is called `inputReady`, $r_t$ is called `inputTransfer`, and so on. We could have annotated the modules `xmas_queue_2` and `xmas_queue_4` in separate statements. However, by using `upperBound` to indicate an integer, we annotate queue modules of all sizes in one `annotate` statement.

We can use the same mechanism to annotate registers:

```
annotate register;
  module(.upperBound(1)) register_mod;
  inputReady !q; // low = ready to rise
  inputTransfer d && !q; // rise
  outputReady q; // high = ready to fall
  outputTransfer !d && q; // fall
endannotate
```

Since we do not look at the implementation of a black box, any registers inside it are hidden as well. This means that to express a property $p : (I \oplus F \to \mathbf{2}) \to \mathbf{2}$, the value of a register $f \in F$ may not be available: we may not be able to determine $\sigma(f)$. Instead, only the inputs $I$, some visible registers $F'$ and all outputs wires of black box modules are available. In the analysis, output wires of black boxes take the role of free variables, just like input wires in the overall design $I$. Instead of complicating our definition of state to account for output wires of black box modules, we simplify it. All wires that play the role of free variables during our analysis, including $I$ and $F$, will be called *fundamental* wires. Let $W$ be the set of fundamental wires: then all inputs $I \subseteq W$, all registers $F \subseteq W$, and the output wires of black box modules are in $W$.

**Definition 2.3** (Circuit, state, property) A *circuit* consists of (a finite set of) registers $F$, and fundamental wires $W$ such that $F \subseteq W$, together with a next-step function $n : (W \to \mathbf{2}) \to (F \to \mathbf{2})$. A *state* $\sigma$ of a circuit is a function $\sigma : W \to \mathbf{2}$ which assigns a value to every input and every register. A *property* $p$ in a circuit is a function which, given a state, produces a value $p : (W \to \mathbf{2}) \to \mathbf{2}$.

We keep track of the state holding elements in $W$, namely $F$, because it helps us to reason about time dependent behavior caused by the next-step function of the circuit, $n$. Some circuits may have a *starting state*, denoted $\sigma_0$. This denotes a state of the circuit after a reset. Given such a state, we can look at time dependent behavior of RTL circuits.

**Definition 2.4** (successor-state, trace) We say that $\sigma' : W \to \mathbf{2}$ is a successor state of $\sigma$ if for all $f \in F$, $n(\sigma)(f) = \sigma'(f)$. In a circuit with next-step function $n$, a sequence of states $\sigma_0, \sigma_1, \ldots$ is called a *trace* if $\sigma_0$ is a starting state, and $\sigma_{i+1}$ is the successor state of $\sigma_i$, for all $i \geq 0$. When the trace is known, we will refer to $\sigma_i$ as the state at time $i$.

Note that by the annotations in this section, $r_r$ is a property for a queue given by the fundamental wire $i$.trdy. Hence $r_r(\sigma) \leftrightarrow \sigma(i.\text{trdy})$. For $t_r$, we cannot express the property in fundamental wires without knowing more about the circuit: $i$.irdy is not a fundamental wire since it is an input wire to a queue. Its value in terms of fundamental wires depends on where the queue is used. Taking the example from Section 2.1.2, we can express $qi$.irdy, $qi$.data and $qo$.trdy in terms of fundamental wires:

$$\begin{aligned} qi.\text{irdy}(\sigma) &= \sigma(\mathsf{source}.\text{irdy}) \wedge !\sigma(\mathsf{source}.\text{data}[0]) \\ qi.\text{data}[0](\sigma) &= \sigma(\mathsf{source}.\text{data}[0]) \\ qo.\text{trdy}(\sigma) &= \sigma(\mathsf{source}.\text{irdy}) \wedge \sigma(\mathsf{source}.\text{data}[0]) \end{aligned}$$

Consequently, we can express $t_{qi}$ and $t_{qo}$ in fundamental wires:

$$\begin{aligned} t_{qi}(\sigma) &= qi.\text{irdy}(\sigma) \wedge r_{qi}(\sigma) = \sigma(\mathsf{source}.\text{irdy}) \wedge !\sigma(\mathsf{source}.\text{data}[0]) \wedge \sigma(qi.\text{trdy}) \\ t_{qo}(\sigma) &= r_{qo}(\sigma) \wedge qo.\text{trdy}(\sigma) = \sigma(qo.\text{irdy}) \wedge \sigma(\mathsf{source}.\text{irdy}) \wedge \sigma(\mathsf{source}.\text{data}[0]) \end{aligned}$$

We can also use the definitions for $r_r$, $r_s$, $t_r$ and $t_s$ for gate level descriptions as well. The next chapter will provide insight in the relation between gate level and RTL.

# Chapter 3
# Analysis of circuits

In the previous section, we saw how NoCs could be described in terms of xMAS. We aim to analyse NoCs that cannot be described in xMAS as well. One of the best known types of interconnect is a bus. In a bus, multiple components can talk to each other. When a wire in a design is used in two directions, called an inout wire, cycles occur. Unlike sequential cycles, where the cycle is intended to store state, such cycles are combinational. Combinational means that, under certain inputs, the value of all wires is determined. That is: the wire value does not depend on timing, or previously held capacitive charges. This chapter shows how to adapt such structures, such that it can be analysed with conventional methods.

There are several strategies to deal with inout wires. A common way to deal with combinational cycles, is to let X denote the value of all wires in such a cycle during analysis. This means that X is used in its interpretation as invalid value, even if the design is valid. Another way to deal with inout wires, is to avoid the cycles from arising. This is done by modeling the inout wire with an input and an output wire, if possible. The approach taken here is to give an interpretation to designs with cycles.

In Verilog, hardware is described using a four-valued logic: 0, 1, X, and Z. A value Z models an open or high impedance wire while a value X, in Verilog, means unknown, invalid, or a don't care. Note that VHDL uses a more fine-grained approach, where X only stands for invalid, and unknown is modeled by $U$. Verilog has no $U$. Another maybe counter intuitive property, is that the output of a `not` gate driven by Z is X, which is also the output of a `not` gate driven by X. Some work on four- on five valued logic, such as that by Bergstra in [Bergstra and Ponse, 1999], has the nice property that $\neg(\neg(v)) = v$ for all $v$, which is not the case in hardware.

To handle cycles in designs, it is recognised that a fixed-point should be calculated. In constructive logic, such a fixed-point can be calculated by starting with absence of knowledge, leading to derived values, as shown by Kees Goossens [Goossens, 1993]. The approach taken here is to model absence of knowledge with a fifth value, $\perp$, as used in work by Riedel and Bruck [Riedel and Bruck, 2012]. Contrary to the work of Riedel, we give an interpretation of circuits in which four-valued logic is also used, allowing us to deal with inout wires. A side-effect of using Verilog's four-valued logic, is that we can also use X as a return value, which

Riedel and Bruck could not. After completing our analysis, should values of $\bot$ remain (meaning we cannot derive their values), we will return $\mathsf{X}$. This allows us to keep $\mathsf{X}$ as a local value, while translating the rest of the circuit, where the work of Riedel and Bruck [Riedel and Bruck, 2012] would just make the observation that the entire circuit is not combinational.

We relate our semantics to circuits with previous results from Riedel and Bruck [Riedel and Bruck, 2012]. Riedel and Bruck argue that the use of cyclic circuits can yield better circuits, and give semantics of such cyclic circuits. For the semantics they propose, they show:

- For any cyclic circuit with $n$ gates, an equivalent acyclic circuit with $n^2$ gates exists.
- A concrete circuit with $n$ gates, for which any equivalent acyclic circuit has $2n$ gates.

For our semantics, we will arrive at these results:

- For any cyclic circuit with $n$ gates, there is an equivalent acyclic circuit with $2n^2$ gates, thus 'losing' a factor 2. If we require – as Riedel and Bruck do – that the wires in the cyclic circuit are always 0 or 1, then for any cyclic circuit with $n$ gates, there is an equivalent acyclic circuit with $n^2$ gates.

This shows that the factor 2 between the acyclic circuits is due to the use of four-valued logic. In addition, we manage to find circuits for which the equivalent acyclic version needs to be much larger:

- We build a concrete circuit with $n$ gates, for which any equivalent acyclic circuit using the same gates has $\frac{1}{2}n(n+1)$ gates. This means that our circuit shows a bigger gap between cyclic and the acyclic circuits.

This is an example of a circuit for which an acyclic equivalent is much larger, when compared to the $2n$ gates required for the circuit provided by Riedel and Bruck.

This chapter will arrive at a procedure that uses five values: $\mathbf{5} = \{0, 1, \mathsf{X}, \mathsf{Z}, \bot\}$. We interpret a gate level circuit $m : \mathbf{5}^W \to \mathbf{5}^W$ as a two-value logic (Boolean) formula $f : \mathbf{2}^I \to \mathbf{2}^O$, and pay some attention to how to do this symbolically. This allows us to interpret gate level circuits as Boolean formulas in the rest of this thesis. We will describe how we interpret a gate level circuit by a function $m : \mathbf{5}^W \to \mathbf{5}^W$ in the next section, and gradually reshape this into the form $f : \mathbf{2}^I \to \mathbf{2}^O$.

## 3.1 Introduction

First, we show what could go wrong when using a straightforward gate translation. Rivest already showed in 1977 that cycles in circuits are required to design minimal circuits [Rivest, 1977]. Our first example will be without cycles, but shows a circuit that contains an error. A straightforward gate-wise translation to SAT would cause us to derive properties that simply do not hold, including properties that do not even concern (this part of) the circuit.

Consider the Verilog code corresponding to the design in Figure 3.1:

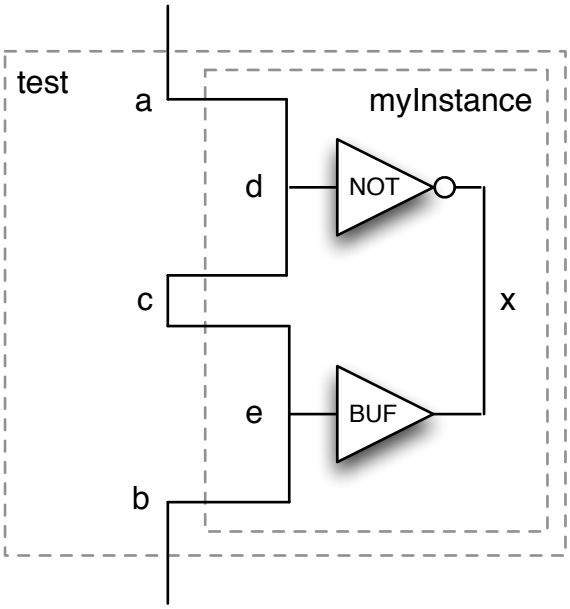



**Figure 3.1:** *A design for which formal analysis is tricky.*

```
module test (a, b)
 input a; output b; wire c;
 submod myInstance (a,c,c,b);
endmodule
module submod (d,d,e,e)
 inout d; inout e; wire x;
 not(x,d); // x = ! d
 buf(x,e); // x = e (information flows from e to x)
endmodule
```

The module interface `(d,d,e,e)` is unconventional. It is a shorthand for the interface `(d,d2,e,e2)`, where `d2` is assigned to `d`, and `e2` to `e`. The industrial parser we work with accepts the Verilog code above as valid input.

Consider what would happen if the module `test` would be given some input, say 0: the wire `a` is connected to 0, giving it the value 0. Module `submod` is instantiated under the name `myInstance`. In this instantiation, it is given value 0 as its first parameter, and it uses the same wire called `d` as its second parameter. Hence the value of wire `a` and that of wire `d` are equal (both 0). Looking at `myInstance` again, that value of `d` is assigned to `c`, which assigns it to `e`, finally causing the output `b` to have the same value. In hardware, this part of the design is just a single wire, which is called `a`, `b`, `c`, `d` and `e` in various places of the code. Note that there are two gates (buf and not) that share one output wire, called x. Since these gates are directional, they do not influence the value of their inputs (d and e). The not gate will return a low output on a high input, and a high output

on a low input. The buf gate returns a high output on a high input, and a low output on a low output.

A straightforward translation to Boolean logic, or SAT – we use the SMT-LIB notation here – would look like this:

```
(AND (= a d)
     (= d c)
     (= c e)
     (= e b)
     (= x (NOT d))
     (= x e))
```

As expected, this allows us to derive `(= a b)`. Unfortunately, it also allows us to derive `(= 0 b)` and `(= 1 b)`, or even `(= 1 0)`. All of these statements are not valid in the hardware. This is due to the fact that the SAT formulation contains a contradiction, whereas the hardware just contains a local short-circuit at `x`. Of course, a short-circuit is usually undesirable in hardware, but this should not cause our formal analysis techniques to derive $0 = 1$. We give a sound translation from digital circuits to SAT formulas.

The main contribution of this chapter is to put a mathematical foundation under the analysis of four valued circuits (tristate buffers, two-way assignments and multiple assignments) and of cycles, whereas other works only treat either one. The novelty lies in a new interpretation of circuits given in Section 3.2. Previous results on creating acyclic circuits from cyclic ones follow directly from this new interpretation, as shown in Section 3.4. Based on this novel definition, we find a method to reduce such circuits to Boolean formulas in Section 3.5. This shows that any formal tool that can analyse properties symbolically, can be extended to the analysis of circuits with inout wires, etc. We do not consider behavior of circuits that relies on timing. The parts of the circuit that do have a state (registers, memory and clock) need to be modeled separately. This makes most formal tools able to handle a larger set of designs, even when they were just made for Boolean formulas initially. For a practical implementation thereof, we refer to the appendix.

## 3.2 Definition of a Circuit

A *gate level circuit* – notation $(M, W)$ – is a set of modules $M$ over a set of wires $W$. Our definition of gate level circuit differs from the (RTL) circuit given in the previous chapter. For notation, we will use $w \in W$ to be a wire. Different wires will be denoted as $w_1$, $w_2$, ... Each wire can hold a value from the set $\{0, 1, \mathsf{X}, \mathsf{Z}, \bot\}$. We write $\mathbf{5}$ for this set: $\mathbf{5} = \{0, 1, \mathsf{X}, \mathsf{Z}, \bot\}$. The value which is not in standard four-valued logic, namely $\bot$, is used to indicate that the wire value is not (yet) known. It is only an intermediate value. According to the Verilog standard, $\mathsf{X}$ can be used to indicate a value that is not stable, so we can get rid of value $\bot$ in final descriptions.

The standard four values stand for: a low voltage drive 0, a high voltage drive 1, an unknown value $\mathsf{X}$, and an undriven value $\mathsf{Z}$. We say that a wire is 'driven' when there is a conductive path between it and either the high or the low voltage.

| *in* \ *out* | $\bot$ | Z | 0 | 1 | X |
|---|---|---|---|---|---|
| $\bot$ | $(\mathsf{Z}, \bot)$ | $(\mathsf{Z}, \mathsf{Z})$ | $(\mathsf{Z}, 0)$ | $(\mathsf{Z}, 1)$ | $(\mathsf{Z}, \mathsf{X})$ |
| Z | $(\mathsf{Z}, \mathsf{X})$ | $(\mathsf{Z}, \mathsf{X})$ | $(\mathsf{Z}, \mathsf{X})$ | $(\mathsf{Z}, \mathsf{X})$ | $(\mathsf{Z}, \mathsf{X})$ |
| 0 | $(0, 1)$ | $(0, \mathsf{X})$ | $(0, \mathsf{X})$ | $(0, 1)$ | $(0, \mathsf{X})$ |
| 1 | $(1, 0)$ | $(1, \mathsf{X})$ | $(1, 0)$ | $(1, \mathsf{X})$ | $(1, \mathsf{X})$ |
| X | $(\mathsf{X}, \mathsf{X})$ | $(\mathsf{X}, \mathsf{X})$ | $(\mathsf{X}, \mathsf{X})$ | $(\mathsf{X}, \mathsf{X})$ | $(\mathsf{X}, \mathsf{X})$ |

**Figure 3.2:** *Truth table for the module* $\texttt{not}(in, out)$.

In our semantics this means that the value is 0, 1 or X. A Z value can move to a 0 or 1, depending on conditions not captured in a gate level model, so it is not driven.

An assignment $\bar{x} : W \to \mathbf{5}$ is a mapping of values to all wires $W$. All such assignments over a set of wires $W$, are written as $\mathbf{5}^W$. For instance, if $w_1$ and $w_2$ are wires with values 0 and Z respectively, $\bar{x} = (0, \mathsf{Z})$ is the assignment, and $(0, \mathsf{Z}) \in \mathbf{5}^{\{w_1, w_2\}} = \{0, 1, \mathsf{X}, \mathsf{Z}, \bot\} \times \{0, 1, \mathsf{X}, \mathsf{Z}, \bot\}$. A module $m \in M$ over a set of wires $W$ is a function $m : \mathbf{5}^W \to \mathbf{5}^W$ that, given an assignment of $W$, yields a new assignment of $W$.

We intend to use modules to model entire circuits, just gates, or even a single transistor. We take the not gate as an example. This example should give a general idea of how we model modules as functions. It is common to call the not gate a gate with one input and one output – which would be a function $\mathbf{5}^{\{in\}} \to \mathbf{5}^{\{out\}}$. We take a different approach, and consider it to be a module on two wires with $W = \{in, out\}$ – a function $m : \mathbf{5}^{\{in,out\}} \to \mathbf{5}^{\{in,out\}}$. Its truth-table is shown in Figure 3.2.

The table in Figure 3.2 is over wires called *in* and *out*. Remember that *in* and *out* are just names, not necessarily indicating the flow of information. We discuss what it means for a wire to be an 'output' in Section 3.3. Each pair in the table is an assignment over $W = \{in, out\}$, to be read as $(in, out)$. For instance $\texttt{not}(\bot, \bot) = (\mathsf{Z}, \bot)$ indicates that if wire *in* is unknown, we cannot compute *out*. This explains the $\bot$ in the second place. We can, however, compute *in*: it is Z. A hardware designer would phrase this as: *in* is not driven by this gate.

On the first wire *in*, this function is the identity if $in \neq \bot$. This corresponds with our intuition of 'input': the gate does not affect its value. The output wire *out* is 'driven' by the gate. Of particular interest are the fixed-point values in the table. That is, values for which $\texttt{not}(in, out) = (in, out)$. They can be seen as possible 'solutions' to a circuit with *only* a not gate module. To ensure that such solutions exist, we restrict the functions that we can consider to be a module.

Not every function from $\mathbf{5}^W$ to $\mathbf{5}^W$ actually represents a module. When a wire has value 1, there is no way to 'unknow' this value. Our not gate respects this. To formalize this, we introduce a partial order $\lesssim$ to indicate which value has the most information. Conflicting information will be modeled as X, no information (value not yet known) as $\bot$. Hence in the partial order $\lesssim$, $\bot$ is the unique smallest element, X is the unique largest, and all other elements are incomparable, as shown in Figure 3.3. For notational convenience, we define the join $\sqcup$ with respect to this order.

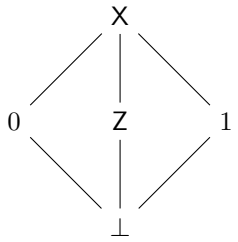



**Figure 3.3:** *Hasse diagram for* $\lesssim$.

**Definition 3.1** ($\lesssim$, $\sqcup$) The partial order $\lesssim$ is defined on $\mathbf{5}$ by:

$$u \lesssim v \Leftrightarrow (u = v \vee u = \bot \vee v = \mathsf{X})$$

We also apply this order on assignments and functions, so for $f, g \in \mathbf{5}^W$ and $u, v \in W$, and for $f, g : \mathbf{5}^W \to \mathbf{5}^W$ and $u, v \in \mathbf{5}^W$; we define:

$$f \lesssim g \Leftrightarrow \forall u.\ f(u) \lesssim g(u)$$

We write $u \sqcup v$ for the least upper bound of $u$ and $v$ with respect to $\lesssim$.

While Definition 3.1 fully defines $\sqcup$ on values, assignments and functions, it might help to see the operation written out. This is its definition on values:

$$u \sqcup v = \begin{cases} \mathsf{X} & \mathsf{X} \in \{u, v\} \vee (u \neq v \wedge \bot \notin \{u, v\}) \\ u & u = v \vee v = \bot \\ v & u = v \vee u = \bot \end{cases}$$

The definition has overlapping cases, but these cases have the same value, so the definition is well-formed.

We require that modules are monotonically increasing with respect to this order. In words, this means that as we learn more about the input values, we should also learn more about the output values.

**Definition 3.2** (monotonically increasing) A function $f : \mathbf{5} \to \mathbf{5}$, $f : \mathbf{5}^W \to \mathbf{5}$ or $f : \mathbf{5}^W \to \mathbf{5}^W$ is monotonically increasing if for all $u$ and $v$ ($u, v \in \mathbf{5}$, $u, v \in \mathbf{5}^W$ or $u, v \in \mathbf{5}^W$ respectively):

$$u \lesssim v \Rightarrow\ f(u) \lesssim f(v)$$

A related requirement is that modules are not allowed to 'forget' information. This property is called 'monotonic'. We require modules to be monotonic.

**Definition 3.3** (monotonic)

$$\bar{x} \lesssim m(\bar{x})$$

As $\sqcup$ is the join for $\lesssim$, this definition is equivalent to:

$$m(\bar{x}) = \bar{x} \sqcup m(\bar{x})$$

To understand what a module does, we will first look at Figure 3.2. Suppose we know nothing about the inputs, then $\mathsf{not}(\bot, \bot) = (\mathsf{Z}, \bot)$ gives some extra information: the first wire, *in*, is not driven by anything, so it will become equal to $\mathsf{Z}$. We can use this extra information, $\mathsf{not}(\mathsf{Z}, \bot) = (\mathsf{Z}, \mathsf{X})$, so the second wire will become $\mathsf{X}$. If we try again, we get $\mathsf{not}(\mathsf{Z}, \mathsf{X}) = (\mathsf{Z}, \mathsf{X})$. This is a fixed-point, the final result for this module. Note that a fixed-point is not necessarily unique: $\mathsf{not}(\mathsf{X}, \mathsf{X}) = (\mathsf{X}, \mathsf{X})$ is also a fixed-point, but it is not the fixed-point we reach from $(\bot, \bot)$. Even a minimal fixed-point with respect to $\lesssim$ is not unique: $\mathsf{not}(0, 1) = (0, 1)$ is incomparable to $(\mathsf{Z}, \mathsf{X})$: both are minimal fixed-points.

We find fixed-points by repeatedly applying a function to its input, increasing the knowledge about the function.

**Definition 3.4** (Kleene-star, $\underline{m}$, $m^*$) Let $\underline{m}$ be given by:

$$\underline{m}(\bar{x}) = \bar{x} \sqcup m(\bar{x})$$

We call $m^*$ the Kleene-star of $m$, which is the limit of $\underline{m}^n$ with $n \to \infty$, or:

$$m^*(\bar{x}) = \bar{x} \sqcup m(\bar{x} \sqcup m(\bar{x} \sqcup m(\cdots)))$$

In this definition, $\underline{m}$ is introduced to 'make $m$ monotonic': the statement '$m$ is monotonic' is equivalent to '$m = \underline{m}$'.

**Lemma 3.1** The Kleene-star of $m$, written $m^*$, is monotonic. If $m$ is monotonically increasing, then so is $m^*$.

*Proof.* Since $\sqcup$ is an upper bound on $\bar{x}$, it follows that $\bar{x} \lesssim \bar{x} \sqcup m(\bar{x}) = \underline{m}(\bar{x})$. We conclude that $\underline{m}^n \lesssim \underline{m}^{n+1}$. So $\underline{m}^n$ forms an increasing sequence in a finite domain. Therefore the limit in Definition 3.4 exists and for some $b$, it holds that: $\underline{m}^n(\bar{x}) = \underline{m}^{n+1}(\bar{x}) = m^*$ if $n \geq b$.

We show that $m^*$ is monotonic. By induction, we show that $\underline{m}^n$ is monotonic. For the base case: $m^0(\bar{x}) = \bar{x}$ is monotonic. We use (in this order) associativity of function composition, the definition of $\underline{m}$, our induction hypothesis, then $\bar{x} = \bar{x} \sqcup \bar{x}$, and again the definition of $\underline{m}$ (with associativity of function composition), to obtain:

$$\begin{aligned}
\underline{m}^{n+1}(\bar{x}) &= \underline{m}(\underline{m}^n(\bar{x})) \\
&= m^n(\bar{x}) \sqcup m(\underline{m}^n(\bar{x})) \\
&= \bar{x} \sqcup m^n(\bar{x}) \sqcup m(\underline{m}^n(\bar{x})) \\
&= \bar{x} \sqcup \bar{x} \sqcup m^n(\bar{x}) \sqcup m(\underline{m}^n(\bar{x})) \\
&= \bar{x} \sqcup \underline{m}^{n+1}(\bar{x})
\end{aligned}$$

Hence $\underline{m}^{n+1}(\bar{x}) = \bar{x} \sqcup \underline{m}^{n+1}$, so $\underline{m}^{n+1}$ is monotonic (for all $n$). By $\underline{m}^{n+1} = m^*$ for some $n$, it follows that $m^*$ is monotonic too.

By associativity of function composition, $\underline{m}^n(\underline{m}^l(\bar{x})) = \underline{m}^{n+l}(\bar{x})$. Let $n$ be a large enough value such that $\underline{m}^n(\bar{x}) = m^*$, then:

$$m^*(m^*(\bar{x})) = \underline{m}^n(\underline{m}^n(\bar{x})) = \underline{m}^{n+n}(\bar{x}) = m^*(\bar{x})$$

Hence $m^*$ yields a fixed-point.

Suppose $m$ is monotonically increasing, then so is $\underline{m}$. For $\bar{x} \lesssim \bar{y}$:

$$\underline{m}(\bar{x})(w) = \bar{x}(w) \sqcup m(\bar{x})(w) \lesssim \bar{y}(w) \sqcup m(\bar{x})(w) \lesssim \bar{y}(w) \sqcup m(\bar{y})(w) = \underline{m}(\bar{y})(w)$$

Therefore $\underline{m}$ is monotonically increasing. The function composition of two monotonically increasing functions also is monotonically increasing, which implies that $\underline{m}^n$ is monotonically increasing if $m$ is. Therefore, we can conclude that $m^*$ is monotonically increasing if $m$ is. □

We can understand $m^*$ in terms of input and output wires. Let $I, O \subseteq W$ be sets of input and output wires, respectively. We define a function $i : \mathbf{5}^I \to \mathbf{5}^W$ as follows:

$$i(\bar{x})(w) = \begin{cases} \bar{x}(w) & \text{if } w \in I \\ \bot & \text{otherwise} \end{cases}$$

Similarly, we can drop wires $o : \mathbf{5}^W \to \mathbf{5}^O$, defined by:

$$o(\bar{x})(w) = \bar{x}(w)$$

With these functions, $o(m^*(i(\bar{x})))$ gives an output assign $\mathbf{5}^O$ for every input assignment $\bar{x} \in \mathbf{5}^I$.

## 3.3 Combining modules

To be able to combine modules, we look at what happens when wires are combined. This is modeled by $u \cdot v$, to indicate that $u$ overwrites $v$. Since there is no inherent order in hardware, this also means that $v$ overwrites $u$: the operator $\cdot$ must be commutative.

This operation can also be applied to assignments, such that assignment $\bar{x}$ overwrites assignment $\bar{y}$, indicated by $\bar{x} \cdot \bar{y}$, obtained by piecewise application of the operation for wires.

**Definition 3.5** (overwrites) We write $\cdot$ for the binary operation *overwrites*. On the set $\mathbf{5}$, $u \cdot v$ it is given by:

| $u$ \ $v$ | $\bot$ | Z | 0 | 1 | X |
|---|---|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| Z | $\bot$ | Z | 0 | 1 | X |
| 0 | $\bot$ | 0 | 0 | X | X |
| 1 | $\bot$ | 1 | X | 1 | X |
| X | $\bot$ | X | X | X | X |

On $\mathbf{5}^W$ and on $\mathbf{5}^W \to \mathbf{5}^W$, operation $\cdot$ is defined as:

$$(\bar{x} \cdot \bar{y})(w) = \bar{x}(w) \cdot \bar{y}(w) \qquad (m_i \cdot m_j)(\bar{x}) = m_i(\bar{x}) \cdot m_j(\bar{x})$$

The definition of *overwrites* corresponds to the Verilog standard for resolving multiple drivers on a wire (Table 6-2 [IEEE, 2001]). The only difference is that we included $\bot$. Since $\bot$ models a lack of information, we choose $\bot$ such that

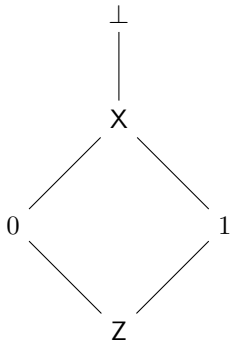



**Figure 3.4:** *Hasse diagram for the poset corresponding to the · operation.*

$v \cdot \bot = \bot$ if the outcome of $v \cdot w$ depends on $w$. The choice for $\bot \cdot \mathsf{X} = \bot$ rather than $\bot \cdot \mathsf{X} = \mathsf{X}$, is made such that $\cdot$ is associative. Choosing $\bot \cdot \mathsf{X} = \mathsf{X}$ would yield $(0 \cdot 1) \cdot \bot = \mathsf{X}$ while $0 \cdot (1 \cdot \bot) = \bot$. It defines a partial order shown in Figure 3.4.

Note that $\sqcup$ and $\cdot$ do not distribute: $0 \sqcup (0 \cdot \mathsf{Z}) \neq (0 \sqcup 0) \cdot (0 \sqcup \mathsf{Z})$. However, we can prove the following:

**Lemma 3.2** Given $u \lesssim v$, then $u \cdot w \lesssim v \cdot w$.

*Proof.* For $w = \bot$, $u \cdot \bot = \bot = v \cdot \bot$: the lemma holds, so assume $w \neq \bot$. For $u = v$ the lemma follows from reflexivity of $\lesssim$, so assume $u \neq v$. We complete our proof with a case distinction on $u \in \{\mathsf{X}, 0, 1, \mathsf{Z}, \bot\}$. Since $u \lesssim v$, $u \neq \mathsf{X}$. If $u \in \{0, 1, \mathsf{Z}\}$, then $v = \mathsf{X}$ and $v \cdot w = \mathsf{X}$, since $w \neq \bot$. If $u = \bot$, then $u \cdot w = \bot$, completing the proof. □

**Lemma 3.3** The operation $\cdot$ is closed for monotonic modules, and for monotonically increasing modules.

*Proof.* From commutativity of $\cdot$ allows us to write Lemma 3.2 as follows:

$$u \lesssim v \Rightarrow w \cdot u \lesssim w \cdot v \tag{1}$$

Combining Lemma 3.2 and the previous equation yields:

$$u \lesssim v \wedge w \lesssim x \Rightarrow u \cdot w \lesssim v \cdot x \tag{2}$$

As a corollary, by filling in $u = w$, and using $u \cdot u = u$:

$$u \lesssim v \wedge u \lesssim x \Rightarrow u \lesssim v \cdot x \tag{3}$$

We can use this to prove that the combination of monotonically increasing functions is monotonically increasing. Suppose $m_1$ and $m_2$ are our monotonically increasing functions. We can perform the following substitution in Equation 2. Let $u = m_1(\bar{x})$, $v = m_1(\bar{y})$, $w = m_2(\bar{x})$ and $x = m_2(\bar{y})$. As a consequence $m_1(\bar{x}) \cdot m_2(\bar{x}) \lesssim m_1(\bar{y}) \cdot m_2(\bar{y})$.

Equation 3 suffices to prove that the combination of monotonic modules will be monotonic. Let $u = \bar{x}$, $v = m_1(\bar{x})$ and $x = m_2(\bar{x})$, then $\bar{x} \lesssim m_1(\bar{x}) \cdot m_2(\bar{x})$. □

The effect that a circuit has on wires is the combined effect of the modules, applied several times. To combine the effect of modules, we can use $\cdot$. We run into a problem if the modules are defined on a different set of wires. For instance, the not gate in Figure 3.2 was defined on $\{in, out\}$. If we want to combine it with, say another not gate, we clearly want to be able to make new combinations. For this, we introduce the function $z$, which can be used when a wire is never driven.

**Definition 3.6** (Not driven, $z$) Function $z$ is defined as:

$$z(w) = \begin{cases} w & \text{if } w \neq \bot \\ \mathsf{Z} & \text{if } w = \bot \end{cases}$$

We say a wire $w$ is *never driven* by $m$ if for all $\bar{x}$:

$$m(\bar{x})(w) = z(\bar{x}(w))$$

Note that Figure 3.2 shows that $in$ is never driven. In fact, like many gates, the not gate has just one non-driven port. We use this as a defining property for 'gate': This means that for every gate $m$, there is at most one wire $w$ that changes its value.

**Definition 3.7** (Gate) An admissible module $m$ on wires $W$ is a *gate with output port* $w$ if:

$$\forall\ \bar{x} \in \mathbf{5}^W,\ v \in W.\quad v \neq w \to m(\bar{x})(v) = z(\bar{x}(v))$$

If it is clear from the context which output port $w$ is intended, or if the intended output port is of no importance, we will say that $m$ is a *gate*.

A module that changes two wires could be modeled as two gates, but never as one. Note that this deviates from the Verilog standard slightly, since in Verilog some gates[1] have multiple outputs.

Using the observation that a not gate has an input and an output port, it can be defined independently of $W$, as long as $W$ contains $in$ and $out$:

$$\mathtt{not}(\bar{x})(w) = \begin{cases} z(\bar{x}(w)) & \text{if } w \neq out \\ \bar{x}(out) \sqcup \mathsf{not}(\bar{x}(in)) & \text{if } w = out \end{cases}$$

where not is given by:

$$\mathsf{not}(x) = \begin{cases} \bot & x = \bot \\ 1 & x = 0 \\ 0 & x = 1 \\ \mathsf{X} & \text{otherwise} \end{cases}$$

Single input function not corresponds exactly to the Verilog specification, with only the $x = \bot$ case added. The way we express the module $\mathtt{not} : W \to W$ can be used as a template for every gate from the specification. Note that the definition of not given above corresponds to the definition given earlier in Figure 3.2. The

[1] An example would be the demultiplexer.

observation that not corresponds to the Verilog specification, relates the Verilog specification to Figure 3.2.

The not gate must be monotonically increasing, so our function not must be, too. Therefore, $\bot$ is the only allowed value for $\mathsf{not}(\bot)$. In general, gates will only return $\bot$ on an output if filling in remaining $\bot$s at the inputs might change that output. This corresponds to 'using all the knowledge we have'. For instance: $\mathsf{and}(0, \bot) = 0$, rather than $\mathsf{and}(0, \bot) = \bot$, as 0 teaches us more than $\bot$ ($\bot \lesssim 0$).

We summarize what we have so far.

- We describe a module as a function $\mathbf{5}^W \to \mathbf{5}^W$.
- To extend the module over a larger $W$, the extra wires behave as $z : \mathbf{5} \to \mathbf{5}$.
- The combination of two modules, $m_1, m_2 : \mathbf{5}^W \to \mathbf{5}^W$, is described as $m_1 \cdot m_2$.
- To compute the final value of a module $m : \mathbf{5}^W \to \mathbf{5}^W$, we use $m^*$, which is a finite computation.

The final value of a combination of modules can be expressed as a single finite computation. That is: a computation that directly yields a fixed-point.

**Definition 3.8** (final value) The modules $(\{m_1, \ldots, m_n\}, W)$ have final value $m : \mathbf{5}^W \to \mathbf{5}^W$, where $m$ is given by:

$$m = (m_1 \cdot \ \cdots \ \cdot m_n)^*$$

The final value $m$ of a set of modules is always monotonic. If all modules are monotonically increasing, so is the final value. This follows from the fact that both $\cdot$ and $^*$ preserve these properties.

As in the previous section, we can find the output values, given a set of input values through $o(m(i(\bar{x})))$. We will show how to turn this into a Boolean formula in Section 3.5. In the next section, we will study the size of the computation given in Definition 3.8. In this analysis, our notion of *gate* will count as the basic unit of computation.

## 3.4 Circuit size

We answer the question: given a circuit with any number of gates, how large does an acyclic circuit need to be to capture the behavior of the original circuit. As pointed out in the introduction, this will improve on the results of Riedel and Bruck [Riedel and Bruck, 2012].

When describing a wire with a Boolean formula, the size of the formula can increase. If we do not use sharing, this increase is exponential in the worst-case, and quite often also in practice. The usual solution is to use Directed Acyclic Graphs (DAGs) to represent formulas. This increases the sharing inside expressions. This applies to the formulas in four-valued logic, as well as the Boolean formulas. A downside to such DAGs, is that they need to be acyclic.

### 3.4.1 Polynomial time algorithm for making a circuit acyclic

Many algorithms have been proposed to change a combinational cyclic circuit into an acyclic one automatically. A combinational cyclic circuit is a cyclic circuit in which no $\perp$ values remain if all the inputs are set to something other than $\perp$.

We found that the algorithms proposed in the literature check that the original circuit is combinational. This is a co-NP-hard problem: checking for cycles is fast, but checking that those cycles are combinational makes the problem hard. This causes such algorithms to have an exponential runtime. This section shows how the semantics defined in this chapter give us an algorithm in a straightforward manner. This algorithm does not take the functionality of the individual gates into account, giving it a polynomial runtime.

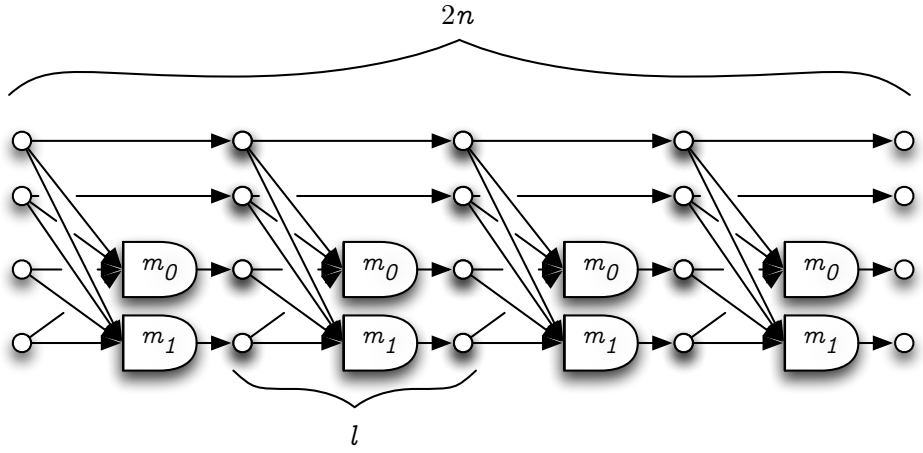



**Figure 3.5:** *Construction of $m^*$: make $2n$ copies of $l$.*

We already defined what we mean by a circuit with $n$ gates, and what it does. Let $l = m_1 \cdot \ \cdots \ \cdot m_n$. So, $l$ is the circuit of our $n$ gates combined, for each gate applied only once. In our definition of the Kleene-star, we already saw that we can use a finite series to express it. For a large enough value of $k$:

$$m = (m_1 \cdot \ \cdots \ \cdot m_n)^* = l^* = \underline{l}^k$$

Since $m_i$ is monotonic, so is $m_1 \cdot \ \cdots \ \cdot m_n$. Therefore, $l = \underline{l}$ and:

$$m = l^k$$

The number of gates used for $m$ is $kn$ ($n$ for each $l$). This way of counting gates still holds if the $\cdot$ operation is to be counted as a gate. In such a case, one $\cdot$ operation is needed in the original circuit for every additional assignment to a wire (adding to $n$). To construct $l$, we use the $\cdot$ operations in exactly the same places, again giving $n$ gates to construct $l$. By our assumption that gates are monotonic, we do not need to introduce a $\sqcup$ operation anywhere.

Since every $m_i$ is a gate, each has an output wire $w_i$. Let $W_o = \{w_1, \ldots, w_n\}$ be this set of output wires. For some input of the network $\bar{x}$, we can give the increasing series:

$$\bar{x}_0 = \bar{x}$$
$$\bar{x}_{j+1} = l(\bar{x}_j) = l^j(\bar{x})$$

Using only that $l$ is monotonic, we conclude that $\bar{x}_j$ is an increasing sequence with respect to $\lesssim$. If $\bar{x}_{j+1} = \bar{x}_j$, the same holds for higher values of $j$. Every wire in $W_o$ can be increased at most twice, from $\bot$ to $\mathsf{X}$ via 0, $\mathsf{Z}$ or 1. Wires in $W_o$ are the only ones that can change. The number of distinct elements in $\{\bar{x}_0, \ldots, \bar{x}_j\}$ is therefore at most twice the number of wires in $W_o$. Since the number of wires in $W_o$ is at most $n$, we conclude that for $k = 2n$, $l^k = m$. That is: $2n$ copies of $l$ suffice to compute $m$. In this case, $2n^2$ gates are used to describe $m$ (without cycles). This makes the algorithm almost trivial (see Algorithm 1). Figure 3.5 graphically illustrates our construction.

**Data**: A set of gates $m_0, \ldots, m_i$
**Result**: A circuit $m$ that is represented by those gates
1 Let $l := m_0 \cdot \ \cdots \ \cdot m_i$;
2 Mutable $m := id$;
3 **for** $2n$ *times* **do**
4 | Mutable $m := l(m)$.
5 **end**

**Algorithm 1:** Algorithm for creating an acyclic circuit.

If we assume that the value $\mathsf{X}$ is never attained, we can repeat the original construction from Riedel and Bruck [Riedel and Bruck, 2012]. If $\bar{x}_j(w) \neq \mathsf{X}$ for all $j$ and $w$, then every wire in $W_o$ is increased at most once. The number of distinct elements in $\{\bar{x}_0, \ldots, \bar{x}_j\}$ is at most the number of wires in $W_o$ in such a case. Hence if $\mathsf{X}$ does not occur, $n$ copies of $l$ suffice to compute $m$. In this case, $n^2$ gates are used to describe $m$. Note that in this construction, not all values for $\bar{x}$ are allowed as input, and for some circuits there is no value for $\bar{x}$ such that $\mathsf{X}$ does not occur in $\{\bar{x}_0, \ldots, \bar{x}_j\}$. In the work of Riedel and Bruck, these circuits are called 'not Boolean' and excluded from their analysis, as well as circuits in which occurrences of $\bot$ remain.

### 3.4.2 A worst-case circuit

The previous analysis gives insight in how to create a worst-case circuit: for every wire, one gate has it as its output wire, and each gate uses every input. We use the full directed graph on $n$ nodes, in which every node is a gate. Such a graph is drawn for $n = 4$ in Figure 3.6. We call the gates in this 'full-graph' circuit $m_1, \ldots, m_n$, and refer to these as nodes.

We do not consider what function the nodes are for now: given an acyclic circuit, and depending on that circuit, we make the nodes perform functions such that, if the output value of the acyclic circuit and our 'full-graph' circuit coincide, the number of nodes in the acyclic circuit is at least $\frac{1}{2}n(n+1)$.

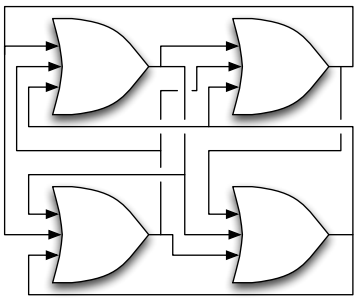


**Figure 3.6:** *A worst-case circuit on $n = 4$ nodes.*

Assume some acyclic circuit uses $k$ gates, called $a_1, \ldots, a_k$. Since the circuit is acyclic, we assume that gate $a_i$ can use values from $a_j$ if $j < i$, but not vice versa. We 'drive' both circuits with $\perp$ for every wire. Circuits $(m_1 \cdot \cdots \cdot m_n)^*$ and $a_1(a_2(\cdots(a_j(\perp, \ldots, \perp))\cdots))$ should give the same outputs on all wires.

Each of these should correspond to one of the original nodes $m_1, \ldots, m_n$. For each node $m_i$, there must be a gate $a_j$ that corresponds to it: if not, let $m_i(\bar{x}) = 1$, and let the other nodes yield 0. If there are wires in the acyclic circuit that correspond to this configuration, they will not correspond to the configuration with all nodes yielding 0, so the acyclic circuit does not perform the same function as the original.

Pick the node $m_i$ for which its first occurrence in $a_1, \ldots, a_k$ is the last among all nodes. Without loss of generality, we assume that this is node $m_n$ (we can rename the nodes if needed). The corresponding gate in $a_1, \ldots, a_k$ is $a_f$ ($f$ for first), with $j \geq n$.

Let node $m_n(\bar{x}) = 1$, and all the other nodes have $\perp$ as output while $m_n$ is not defined. As a consequence, the input for the network $a_i$ with $i > f$ is the same as that with the gates $a_1, \ldots, a_{i-1}$ removed. Similarly, if we remove node $m_n$ and drive its output wire with 1, we are left with a new 'full-graph' on $n - 1$ nodes. This gives two new networks: a new original with $n - 1$ gates, and a new acyclic one with at most $k - n$ gates. In it, we can choose another 'initial' node by letting its value depend on the wire from gate $m_n$. Repeating the process gives $n + (n - 1) + \cdots + 1 = \frac{1}{2}n(n + 1) \geq k$.

There is another – perhaps more intuitive – way to view this construction: for every path through the full graph, there must be a path through the same nodes in the DAG. We construct the gates such that only that path through the original full graph yields 1, and any deviation from it will yield 0 or $\perp$.

A reader may not like that we change the implementation of a gate after the acyclic network is defined. This can be circumvented by adding extra input wires, which can then be used to choose the implementation of our gate.

## 3.5 From gates to Boolean formulas

Circuits are translated to Boolean formulas in two steps. First, all modules are expressed symbolically as formulas in four-valued logic. These four-valued formulas are then encoded in Boolean formulas, such that Boolean tools like SAT solvers can be applied to modules.

To go from $\mathbf{5}$ to four-valued logic, we identify $\bot$ with $\mathsf{X}$. As a consequence, gates that used to be monotonic and monotonically increasing no longer are, and we loose the argument on why we will reach a fixed-point. The rationale behind this is as follows: we may first compute a fixed-point, and then identify $\bot$ with $\mathsf{X}$, such that $\mathsf{X}$ takes its traditional role of being the 'undefined' value. If all gates agree[2] on $\bot$ and $\mathsf{X}$, then first identifying $\bot$ with $\mathsf{X}$, and then computing the fixed-point, will produce the same result. In such a procedure, $\mathsf{X}$ will be the initial value (taking the place of $\bot$) for 'undefined' wires. While setting $\bot$ to $\mathsf{X}$ or setting $\mathsf{X}$ to $\bot$ is merely a syntactical difference, we choose the former as $\mathsf{X}$ is a Verilog value, while $\bot$ is not. We did not need the $\sqcup$ operation in computing the fixed-point, we only need to define $\cdot$ on $\mathbf{4} = \{0, 1, \mathsf{X}, \mathsf{Z}\}$. We can reuse its original definition.

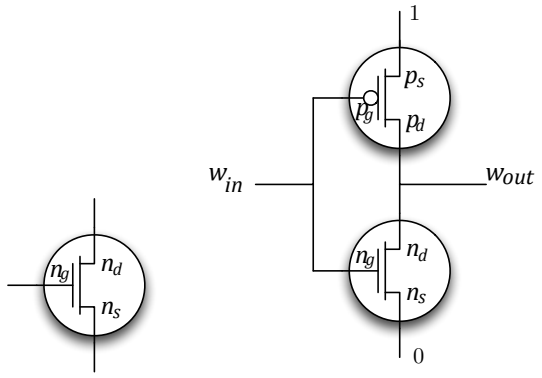
To illustrate our approach and the modeling of inout wires, we consider an n-type Metal-Oxide-Semiconductor field effect transistor (nMOS) transistor (see Figure 3.7(a)). This transistor has three wires: source ($n_s$), drain ($n_d$) and gate ($n_g$). If the input to the gate is high, the drain is connected to the source. If the input to the gate is low, the drain acts like an open wire. We also consider a p-type Metal-Oxide-Semiconductor field effect transistor (pMOS) transistor that connects the drain to the source when the input gate is low. Figure 3.7(b) shows a CMOS inverted composed of an nMOS and a pMOS transistors. We will describe the circuit using two helper functions, $\mathsf{nMOS} : \mathbf{4} \times \mathbf{4} \to \mathbf{4}$ and $\mathsf{pMOS} : \mathbf{4} \times \mathbf{4} \to \mathbf{4}$. Function $\mathsf{nMOS}(g, x)$ equals $x$ whenever $g = 1$, and it equals $\mathsf{Z}$ when $g = 0$, as in Figure 3.7(c). Function $\mathsf{pMOS}(g, x)$ equals $x$ whenever $g = 0$, and it equals $\mathsf{Z}$ when $g = 1$.

With respect to the Verilog specification, the specified values L and H have been replaced by $\mathsf{X}$. This safely changes $\mathsf{nMOS}$ into one in which all values are defined. In fact, the Verilog standard considers the $\mathsf{nmos}$ gate to be uni-directional. We could model this as well, but instead focus on modeling the actual transistors, which are symmetrical in silicon. The corresponding Verilog gate modeled here is called 'tranif1' for $\mathsf{nMOS}$, and 'tranif0' for $\mathsf{pMOS}$.

**Modules to four-valued logic** Each module is described using gates. A gate is a function over a set of four-valued arguments that yields a four-valued answer. Our syntax for formulas $\mathsf{F}$ uses function symbols indicating unary or binary gates ($f_1 : \mathbf{4} \to \mathbf{4}$, $f_2 : \mathbf{4} \times \mathbf{4} \to \mathbf{4}$). Typically, function symbols will include the usual gates, e.g. `not`, `nand`. The syntax can of course be extended with any other gates. The mapping assigning formulas to the wires they represent is noted $\mathsf{M}$.

$$\mathsf{F} := w \mid \mathsf{Z} \mid 0 \mid 1 \mid \mathsf{X} \mid \mathsf{F} \cdot \mathsf{F} \mid f_1(\mathsf{F}) \mid f_2(\mathsf{F}, \mathsf{F})$$
$$\mathsf{M} := w \mapsto \mathsf{F};$$

[2] We did not come across any gates in the Verilog standard that do not agree on $\bot$ and $\mathsf{X}$.




(a) nMOS transistor

(b) CMOS inverter

(c) Function $\mathsf{nMOS}(g, x)$

| $g$ \ $x$ | Z | 0 | 1 | X |
|---|---|---|---|---|
| Z | Z | X | X | X |
| 0 | Z | Z | Z | Z |
| 1 | Z | 0 | 1 | X |
| X | Z | X | X | X |

**Figure 3.7:** *Example module and circuit.*

In this syntax, the nMOS transistor of our example is described as follows:

$$n_g \mapsto \mathsf{Z}; \qquad n_s \mapsto \mathsf{nMOS}(n_g, n_d); \qquad n_d \mapsto \mathsf{nMOS}(n_g, n_s);$$

The symbolic $\cdot$ is interpreted as the $\cdot$ in Definition 3.5. This allows us to evaluate every formula $\mathsf{F}$ for some assignment, and understand a set of lines in the syntax of $\mathsf{M}$ as a module $m$. We write $eval(\mathsf{F}, \bar{x})$ for such an evaluation of $\mathsf{F}$.

**Definition 3.9** (Symbolically defined module) A *symbolically defined module* $M$ is a set whose elements have the form $\mathsf{M}$. Function $m'(M, \bar{x}) : W \to \mathbf{4}$ is recursively defined as:

$$m'(M, \bar{x})(w) = \begin{cases} eval(\mathsf{F}, \bar{x}) \cdot m'(M \backslash_{w \mapsto \mathsf{F}}, \bar{x})(w) & \text{if } w \mapsto \mathsf{F} \in M \\ eval(w, \bar{x}) & \text{otherwise} \end{cases}$$

The semantics of $M$ is the module $m : \mathbf{4}^W \to \mathbf{4}^W$ given by $m(\bar{x}) = (m'(M, \bar{x}))^*$.

To give a symbolic representation for the inverter, we simply combine existing modules. We apply two minor optimizations to the symbolic representation, justified by Definition 3.8. (1) In the description of the nMOS, we may omit the line $n_g \mapsto \mathsf{Z}$. (2) The single statement $w_{out} \mapsto n_d \cdot p_d$ has the same semantics as two separate statements for $w_{out} \mapsto n_d$ and $w_{out} \mapsto p_d$ written separately, so we may write the former. The CMOS inverter can thus be modeled as follows:

$$\begin{aligned} n_s &\mapsto \mathsf{nMOS}(n_g, n_d) \cdot 0; & n_d &\mapsto \mathsf{nMOS}(n_g, n_s) \cdot w_{out}; \\ p_s &\mapsto \mathsf{pMOS}(p_g, p_d) \cdot 1; & p_d &\mapsto \mathsf{pMOS}(p_g, p_s) \cdot w_{out}; \\ p_g &\mapsto w_{in}; \quad n_g \mapsto w_{in}; & w_{out} &\mapsto n_d \cdot p_d; \quad w_{in} \mapsto p_g \cdot n_g; \end{aligned}$$

As a simplification procedure, we eliminate unexposed wires. We are interested in the solution to the system constrained under a set of 'inputs'. In our example, we are interested in the module when $w_{in}$ is fixed. Furthermore, we are only

interested in the value of $w_{out}$. This means we seek a solution in which all other wires $(n_s, n_d, \ldots, p_g)$ have their minimal value.

To eliminate a rule $w \mapsto \mathsf{F}$ (if wire $w$ has no rule, use $w \mapsto \mathsf{Z}$), we find a fixed-point (in $w$) for $\mathsf{F}$. To replace all occurrences of $w$ in $\mathsf{F}$ by $x$, we write $\mathsf{F}[w : x]$. The fixed-point in $x$, namely $\mathsf{F}^*[w : x]$, is equivalent to $\mathsf{F}' = \mathsf{F}[w : \mathsf{F}[w : x]]$. To see this: $\mathsf{F}[w : x]$ evaluates to $\mathsf{Z}$, 0, 1 or $\mathsf{X}$. The latter implies that $\mathsf{F}[w : \mathsf{X}] = \mathsf{X}$, so it is always a fixed-point (and $\mathsf{F}' = X$). If $\mathsf{F}[w : x] = \mathsf{Z}$, then $x = \mathsf{Z}$ by admissibility of $\mathsf{F}$ and $\mathsf{Z}$ is a fixed-point. If $\mathsf{F}[w : x] = 0$ then $\mathsf{F}' \neq \mathsf{Z}$ (again by admissibility) so either $\mathsf{F}' = 0$, in which case 0 is a fixed-point, or $\mathsf{F}' = \mathsf{X}$, which is always a fixed-point. Should we compute a fixed-point from $\mathsf{X}$, then $\mathsf{F}^*[w : \mathsf{X}] = \mathsf{F}[w : \mathsf{X}]$, which is significantly easier to compute.

We apply this procedure to our CMOS inverter example by calculating a fixed-point in $\mathsf{Z}$ for the non-input wires, and $w_{in}$ for the input wire. We first eliminate $n_s$. This yields $n_d \mapsto \mathsf{nMOS}(n_g, \mathsf{nMOS}(n_g, n_d) \cdot 0) \cdot w_{out};$. The fixed-point for $n_d$ can be simplified using: $\mathsf{nMOS}(n_g, \mathsf{Z}) \cdot 0 = \mathsf{Z} \cdot 0 = 0$ and $\mathsf{nMOS}(n_g, \mathsf{nMOS}(n_g, x) \cdot y) = \mathsf{nMOS}(n_g, x \cdot y)$. This yields $\mathsf{nMOS}(n_g, 0 \cdot w_{out} \cdot 0) \cdot w_{out} = \mathsf{nMOS}(n_g, 0 \cdot w_{out}) \cdot w_{out}$. Doing the same for $n_p$ allows us to complete the procedure. We can omit $w_{in} \mapsto w_{in} \cdot w_{in}$ and are left with:

$$w_{out} \mapsto \mathsf{nMOS}(w_{in}, 0 \cdot w_{out}) \cdot w_{out} \cdot \mathsf{pMOS}(w_{in}, 1 \cdot w_{out}) \cdot w_{out};$$

Finding the least fixed-point of $w_{out}$ simplifies things a little:

$$w_{out} \mapsto \mathsf{nMOS}(w_{in}, 0) \cdot \mathsf{pMOS}(w_{in}, 1); \tag{4}$$

In this case, this is the unique minimal solution for our circuit (given a value for $w_{in}$). In general, each wire will either have a minimal value, or be equal to $\mathsf{X}$.

Note that every time we simplify our design by eliminating a single wire, we potentially double its size. The replacement $\mathsf{F}' = \mathsf{F}[w : \mathsf{F}[w : \mathsf{Z}]]$ requires two copies of $\mathsf{F}$: one where $w$ is replaced by $\mathsf{Z}$, and one where it is replaced by $\mathsf{F}[w : \mathsf{Z}]$. After $n$ replacements, we could increase the size of a network by a factor of $2^n$. By using the construction illustrated in Section 3.4.1, we can prevent this blowup.

**Four-valued logic to SAT** Formulas in four-valued logic can be changed into Boolean formulas. To allow for shared expressions in some gates, it is beneficial to use a fresh variable for each wire to indicate which bit we are interested in. Therefore, each wire is encoded as two bits.

For each wire $w$, we write $w'$ for the fresh variable introduced for $w$. The value of wire $w$ can then be encoded as a Boolean function $f(w')$. To get back the value in the four-valued logic, we use the following Boolean lookup table:

$$w : \begin{cases} \mathsf{Z} & \text{if } f(0) = 0 \text{ and } f(1) = 1 \quad (f \text{ is the identity on } w') \\ 0 & \text{if } f(0) = 0 \text{ and } f(1) = 0 \quad (f = 0, \text{ independently of } w') \\ 1 & \text{if } f(0) = 1 \text{ and } f(1) = 1 \quad (f = 1, \text{ independently of } w') \\ \mathsf{X} & \text{if } f(0) = 1 \text{ and } f(1) = 0 \quad (\text{otherwise}) \end{cases}$$

This means that for every wire $W$, we introduce a second wire.

Other choices are possible. The advantage of this choice is that we can encode 0 as 0, and 1 as 1, without introducing helper variable $w'$. The Boolean encoding of each gate is found through its truth-table. We give the Boolean encoding (which uses an if-then-else for the case distinction) of $\cdot$ as an example:

$$(f_1 \cdot f_2)(w') = \begin{cases} f_1(w') \vee f_2(w') & \text{if } w' = 0 \\ f_1(w') \wedge f_2(w') & \text{if } w' = 1 \end{cases} \tag{5}$$

To see what happened with this second choice, let's assume that for some network, all inputs are either 0 or 1. Using two bits per value in four-valued logic, our module is given by a signature of type $w^I \to w^{O+O}$. With $O+O$ we duplicated every formula for an output wire. Every output wire $o$ is defined by two values: $o_1, o_2 \in O + O$. By using an extra helper variable $w'$, we can distinguish between the two: if $w'$ is low, return $o_1$, if $w'$ is high, return $o_2$. This gives a signature of type $w^{I'} \to w^O$, where $I' = I \cup \{w'\}$. If an output value does not depend on $w'$, it represents a 0 or a 1 value. Otherwise, it represents a Z or X value.

If it is clear that a wire will only hold a 0 or 1 value, it can be encoded with just a single SAT variable instead of two. This consideration was already made by Drechsler et al. [Drechsler et al., 2008] and applies to our setting as well. In our example, we used gates that are hardly ever used directly, immediately introducing cycles and Z values. In most hardware designs, the majority of the number of gates used returns a 0 or 1 whenever the all inputs are in $\{0, 1\}$. This corresponds to a value being independent from $w'$, which means we do not need to consider $w'$ as an input value. In all examples we considered, simplifying expressions in four valued logic was sufficient to eliminate all $\cdot$ operations and most Z and X values. Among these designs were cross-bars that used tristate buffers. This was to be expected, since designers wish to avoid making errors that are due to incorrect handling of Z values or due to cycles. Tools usually warn whenever a $\cdot$ cannot be eliminated (through warnings about 'writing to an input' or 'double assignment').

## 3.6 Discussion

Dealing with X and Z values remains an active issue. Several workarounds have been proposed in the past and recently. When using value X in a specification, the intention is a don't care. Synthesizers and test generators [Drechsler et al., 2008] can ignore value X in case of output wires with value X. In implementations, this value should not arise, thus X means here 'invalid'. Some tools yield an X when unable to predict a value, giving it the meaning 'unknown'. In related work, designers are simply warned against the many pitfalls of X [Mills, 2012; Sutherland and Mills, 2007; Turpin, 2003]. Nevertheless, a formal semantics of four valued logic exist, for example the `4v` books in ACL2 [Kaufmann et al., 2015]. This semantic does not consider circuits with cycles. It also uses a different condition for monotonically increasing: in the partial order `4v-<=`, X is the unique smallest element, and all other elements are incomparable. As a result, all their gates are monotonically increasing: a gate that is admissible and monotonically increasing with respect to our partial order $\lesssim$ is monotonically increasing in the sense defined with respect to `4v-<=`.

The intermediate value $\perp$ is also used by Riedel and Bruck [Riedel and Bruck, 2012]. They use the same ordering, and assume monotonic gates, although they do not make this requirement explicit. The element $\perp$ is used in the way we do here. They do not use the values $\mathsf{X}$ and $\mathsf{Z}$. We have shown how to extend their approach to four-valued logic.

Encoding four-valued logic inside two-valued logic has been suggested by Hunt and Swords [Hunt and Swords, 2009]. In this work, two Booleans are encoded per wire instead of just one. One of these is called 'offset', the other 'onset'. We propose using a single free variable for this purpose, such that only one symbolic Boolean value per wire is used. Our lookup table also differs from theirs. These differences are not fundamental, it just shows that the distinction between four values can be encoded in several ways.

Backes and Riedel [Backes et al., 2008; Riedel and Bruck, 2012; Backes and Riedel, 2012] and Bruck [Riedel and Bruck, 2007] consider cycles in stateless circuits. When formal methods are applied to such circuits, they use equivalent acyclic circuits, which can be created by breaking the cycles [Gupta and Selvidge, 2012; Riedel and Bruck, 2012]. In the work of Stephen Edwards [Edwards, 2003], an algorithm is proposed for this. In the paper, Edwards acknowledges that his algorithm has an exponential runtime, but that the growth is slow in practice. Nevertheless, a later paper by Neiroukh, Edwards et al. [Neiroukh et al., 2008] gives another exponentially complex algorithm for removing cycles that is supposed to fix the exponential blowup experienced in the previous paper. In the paper by Backes et al. [Backes and Riedel, 2012], a proof is made that for every combinational cyclic circuit with $m$ gates, there is an acyclic circuit with $m^2$ gates.

Even though we propose different semantics, our goal is not to be more general than all existing related work. Indeed: works that consider the full temporal behaviour of a circuit, as in asynchronous analysis [Eggersglüß et al., 2010], are more general than our work, and significantly more complex.

## 3.7 Conclusions

Typical hardware includes cycles, open wires, inout wires, and multiple assignments. These features make the use of formal verification techniques difficult. We proposed a novel definition of such *circuits*. This definition handles circuits with cycles and four valued logic.

As we can describe more circuits, we were able to find cyclic circuits with $n$ gates, for which the acyclic counterpart contains at least $\frac{1}{2}n(n+1)$ gates. This improves on previous results by Riedel and Bruck, who showed size differences of only a constant factor.

Finally, we discuss how a translation can be made from such circuits to those that are readable by conventional tools. This is done through a reduction of such circuits to Boolean formulas. This enables the application of Boolean verification techniques – like SAT – to more hardware designs.

Since we can move from gate level Verilog to Boolean functions, the rest of this thesis will focus on circuits in which the interpretation of the combinational part of a circuit $m : \mathbf{5}^W \to \mathbf{5}^W$ is known. We can also assume that it satisfies $m(m(\bar{x})) = m(\bar{x})$, that is: it reaches a fixed point in one iteration. In case the

sets of in- and output wires $I, O \subseteq W$ are known, we can fill in $\bot$ for unassigned wires, and the values 0 and 1 for the others, giving us a function of type $\mathbf{2}^I \to \mathbf{5}^O$. Identifying $\bot$ with $\mathsf{X}$ and using an additional variable to represent all four values as Booleans, and adding an additional element to $I$: $I' = I + 1$, we get a function of type $\mathbf{2}^{I'} \to \mathbf{2}^O$.

# Chapter 4
# Invariants

This chapter presents a method to extract inductive invariants from the RTL design automatically. High-level invariants are key to rule-out false deadlocks or other spurious counter-examples in progress verification. Ray and Brayton [Ray and Brayton, 2012] write:

> The fact that buffer relations are making proofs inductive for many designs is the key factor that makes our approach scalable.

Ray and Brayton use the word 'buffer' to refer to a 'queue', and by 'buffer relations', they refer to invariants.

> These seemingly intuitive relations are quite non-trivial to mine from a bit-level implementation of a fabric, unless they are explicitly hinted by the architects.

Invariants are either added manually or automatically extracted from xMAS representations of the fabrics.

This illustrates the aim of this chapter: make model checking approaches scalable, by mining invariants from a bit-level implementation of the fabric. A verification engineer would use this approach to add assumptions about his model. Although the invariants generated in this chapter all hold by construction, they turn out to be fast to check as well.

Provided well-defined interfaces of queues and registers, as described in Chapter 2, the method is automatic and scales to large fabrics. We decompose designs into a set of queues and a set of registers (flip-flops), all interconnected by combinational logic. The queue interface precisely defines when a packet enters and when a packet leaves a queue. From this definition, invariants express the number of packets stored in a queue as the difference between the number of packets, which have entered a queue and the number of packets, which have left it. The main components of our approach are a symbolic function computing the number of times a wire has been high until the current time and the translation of the usual Boolean connectives to expressions over this function.

## 4.1 Method

To obtain invariants, we focus on the interfaces of queues. For this chapter, the focus on queues also allows us to compare our results with invariants that have been added manually. We require designers to indicate which Verilog modules are queues. For each queue, we generate an equality about the number of packets in that queue. Each equality states that a variable representing the number of packets in a queue, must be equal to a sum of linearly independent variables. This sum is obtained by rewriting Boolean expressions about the transfer of packets into linear expressions.

The final step, which produces the invariants, is Gaussian elimination on the system of equalities. This step is the same in many methods for deriving invariants, such as those for Petri-Nets and for xMAS networks [Chatterjee and Kishinevsky, 2012]. Our focus is on finding the system of equalities. We assume the reader is familiar with Gaussian elimination and systems of linear equalities.

**Definition 4.1** (Linear) We say that an equation is linear if it is of the form:

$$c_1 v_1 + c_2 v_2 + \cdots = c_0$$

where the $c_i$ are constants, and $v_i$ are variables. Except for $c_0$, the constants $c_i$ are nonzero.

We start with an example to illustrate the invariants we generate. We show in Section 4.1.1 that it suffices to know equalities about a cumulative function called $s$. In Section 4.1.4, we zoom in on the properties of $s$ essential to obtaining these equalities. We finish with an overview of the algorithm in Section 4.1.5.

### 4.1.1 A simple example

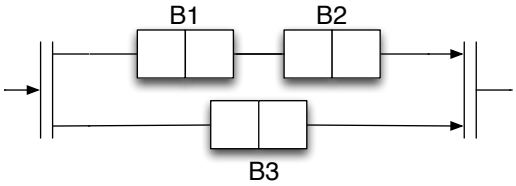



**Figure 4.1:** *A very simple network with three queues, in which one invariant holds.*

Figure 4.1 shows a micro-architectural model of a simple network. It is a network with one input and one output. At the input, a *fork* element duplicates packets, which are offered to queues B1 and B3. Queue B1 is connected to B2, and the packets in queues B2 and B3 are combined at the output by a *join* element. From the RTL implementation of this network, we automatically derive the following invariant:

$$num(\mathsf{B1}) + num(\mathsf{B2}) = num(\mathsf{B3})$$

Such an invariant seems obvious from the micro-architectural model. It can indeed be automatically inferred from such a model. The challenge is that this nice architectural structure is not directly available from the RTL design.

The basic idea of our method is to define variables $enter(x)$ and $exit(x)$. Variable $enter(x)$ represents the number of packets that have entered queue $x$. Variable $exit(x)$ represents the number of packets that have left queue $x$. We assume that at time 0, there is a reset and all queues are empty. Therefore, the number of packets currently in queue $x$ up to some global time $N$ is expressed as the difference between $enter(x)$ and $exit(x)$.

Regarding the example in Figure 4.1, we can express the following equalities:

$$\begin{aligned} num(\mathsf{B1}) &= enter(\mathsf{B1}) - exit(\mathsf{B1}) \\ num(\mathsf{B2}) &= enter(\mathsf{B2}) - exit(\mathsf{B2}) \\ num(\mathsf{B3}) &= enter(\mathsf{B3}) - exit(\mathsf{B3}) \end{aligned}$$

The three equalities above are simplified by performing a translation of $enter(x)$ and $exit(x)$. After this translation, we perform a Gaussian elimination to eliminate all variables except for $num(x)$. A key aspect of the translation is to identify the following equalities, where $v_1$, $v_2$ and $v_3$ are arbitrary variables:

$$\begin{aligned} enter(\mathsf{B1}) &= v_1 \\ enter(\mathsf{B2}) &= v_2 \\ enter(\mathsf{B3}) &= v_1 \\ exit(\mathsf{B1}) &= v_2 \\ exit(\mathsf{B2}) &= v_3 \\ exit(\mathsf{B3}) &= v_3 \end{aligned}$$

These equalities play a central role in the derivation of the invariants: using Gaussian elimination, we can derive the aforementioned invariant. We will see how we got to the variables $v_1$ to $v_3$ in the next sections.

### 4.1.2 Well-defined interfaces

In order to show soundness of our approach, we give an interpretation of $enter(x)$ and $exit(x)$. In our interpretation, the integer $N$ is used, which stands a the time period in terms of clock ticks. Our interpretation of $enter(x)$ and $exit(x)$ will be determined by the number of events when a packet is entering or leaving queue $x$ during that period. For every queue an expression can be defined to indicate when a packet is entering, and another one to indicate when one is leaving a queue. Providing these expressions is the only manual requirement for the RTL designer. The rest of our method is fully automatic.

Our method is fully transparent to the number of places in a queue. We only look at data bits that are used for routing, so if a fixed number of bits is used for routing, our method is also transparent in the width of the queues.

The expressions that determine when a packet is entering, and when one is leaving, have been described in Section 2.2.1. The property $t_r$ describes when a

packet is entering, while $t_s$ describes when one leaves. To relate variables $enter(x)$ and $exit(x)$ with properties, we use an event counter – noted function $s$ – as follows:

$$enter(x) = s(t_r, N)$$
$$exit(x) = s(t_s, N)$$

In Section 4.2 we give an example of what $s$ might be.

The variable $N$ can be seen as a free integer constant. The properties $t_r$ and $t_s$ are described symbolically as Boolean expressions. Our method requires the definition of these events for all queues of the design. Note that these definitions need not be the same for all queues.

### 4.1.3 Interpretation of function $s$

The bit-level structure of the network is known in terms of values such as input wires and queue interface wires, and gates such as `and`, `or`, `xor` and `not`. The value of a wire $w$ at time $t$ is given by function $v(w, t)$, which returns 0 for false (low) and 1 for true (high). Time is indicated by natural numbers (starting at time 0). To obtain the number of 1 values of some wire $w$ up to some time $t$, we can accumulate $v$ to obtain function $s$:

$$s(w, t) = \sum_{i=0}^{t-1} v(w, i)$$

Keeping this interpretation of $s$ in mind, allows the reader to verify the soundness of the translation of $s$.

### 4.1.4 Translation of function $s$

In this section, we translate $s(x, N)$ into a set of variables. We show in Theorem 4.1 (see next sub-section) that the variables used at the end of the translation are linearly independent. Taking a variable for every occurence $s(x, N)$ does not suffice to obtain the correct inequalities, since it would not identify $s(x, N) = s(\neg\neg x, N)$ by itself. That is: $s(x, N)$ and $s(\neg\neg x, N)$ are still linearly dependent. The proposed translation does identify equivalent terms.

In this Section, we focus on properties of $v$. Given these properties, we define a rewrite system on $s$, allowing us to omit $v$.

The reader may verify that:

$$v(0, t) = 0$$
$$v(\neg x, t) = v(1, t) - v(x, t)$$
$$v(x \vee y, t) = v(x, t) + v(y, t) - v(x \wedge y, t)$$
$$v(x \text{ XOR } y, t) = v(x, t) + v(y, t) - 2 \cdot v(x \wedge y, t)$$

We are dealing with integers, so $a - b$ is a shorthand for $a + -1 \cdot b$. Note that we wrote $v(1, t)$ instead of 1 in the equation for $\neg x$. As a consequence, all the equations are linear without constants. To translate `and` gates (or $\wedge$), we introduce operation

$\otimes$. On $v$, this operation is just multiplication. The operation $\otimes$ is commutative and associative, and the following equations hold:

$$\begin{aligned} v(x \wedge y, t) &= v(x, t) \otimes v(y, t) \\ (a + b) \otimes c &= a \otimes c + b \otimes c \\ 0 \otimes b &= 0 \\ b \otimes b &= b \end{aligned}$$

Note that these equalities applied from left to right, together with associativity and commutativity, translate any Boolean expression into a sum in which every term is a constant times a list $v(a_0, t) \otimes \cdots \otimes v(a_n, t)$. Up to associativity and commutativity of both $+$ and $\otimes$, this translation yields a canonical form.

Lists separated by $\otimes$ can be thought of as sets of constants (without duplicates, since $v(x, t) \otimes v(x, t) = v(x, t)$). Suppose $x$ is some expression translated this way:

$$v(x, t) = \alpha v(a_0 \wedge a_1 \wedge \ldots, t) + \beta v(b_0 \wedge b_1 \wedge \ldots, t) + \cdots$$

By summing over all values of $t$ from 0 to $N$, we obtain:

$$s(x, N) = \alpha s(a_0 \wedge a_1 \wedge \ldots, N) + \beta s(b_0 \wedge b_1 \wedge \ldots, N) + \cdots$$

Hence, for every remaining term $s(c, N)$, $c$ is a conjunct of independent wires. We refer to these final terms as *primitives*. These primitives are the variables in our final Gaussian elimination.

Instead of writing $s(\ldots, N)$, we write the corresponding set of independent variables:

$$s(x, N) = \alpha\{a_0, a_1, \ldots\} + \beta\{b_0, b_1, \ldots\} + \cdots$$

Note that the above is a linear equality, with sets in the places where you would normally expect the variables. The empty set will correspond to value $N$.

We can apply this translation directly on $s$. Where previously $\otimes$ could be thought of as a product, it does not correspond to an operation that can be expressed in terms of $s$. Therefore the operation $\otimes$ must be read as a purely syntactic intermediary. The translation is still sound due to its counterpart on $v$.

Our rewrite system is as follows:

$$\begin{aligned} s(0, N) &= 0 \\ s(1, N) &= \{\} \\ s(\neg x, N) &= \{\} - s(x, N) \\ s(x \vee y, N) &= s(x, N) + s(y, N) - s(x, N) \otimes s(y, N) \\ s(x \text{ XOR } y, N) &= s(x, N) + s(y, N) - 2 \cdot s(x, N) \otimes s(y, N) \\ s(x \wedge y, N) &= s(x, N) \otimes s(y, N) \\ (a + b) \otimes c &= a \otimes c + b \otimes c \\ c \otimes (a + b) &= a \otimes c + b \otimes c \end{aligned} \tag{1}$$

When we are done rewriting in this way, we replace $\otimes$ by interpreting $s(x, N)$ as $1 \cdot \{x\}$ and applying:

$$c_1 \cdot S \otimes c_2 \cdot T = c_1 \cdot c_2 \cdot (S \cup T)$$

To summarize our approach, we look at a small example, the translation of the term $\neg(\neg x \wedge \neg y)$:

$$\begin{aligned} s(\neg(\neg x \wedge \neg y), N) &= 1 \cdot \{\} - s(\neg x \wedge \neg y, N) \\ &= \{\} - (\{\} - \{x\}) \otimes (\{\} - \{y\}) \end{aligned}$$

Since $\{\} \cup \{\} = \{\}$, $\{\} \cup \{x\} = \{x\}$ and so forth:

$$\begin{aligned} &= \{\} - \{\} + \{y\} + \{x\} - \{x, y\} \\ &= \{y\} + \{x\} - \{x, y\} \end{aligned}$$

From the above, one can verify that the translation correctly identifies $s(x \vee y, N)$ with its De Morgan dual.

The translation given by our rewrite system (1) is exponential in terms of the depth of the logical formula, which is bounded in most practical applications.

### 4.1.5 An algorithm for finding inductive invariants

Rewrite system (1) allows us to express $enter(x)$ and $exit(x)$. We can define $num(x)$ directly:

$$num(x) = enter(x) - exit(x)$$

This yields one equation per queue. Note that $num(x)$ denotes the number of packets in a queue *at* time $N$, while $enter(x)$ and $exit(x)$ denote the number of enter and exit events *up to* (not including) time $N$. All linear inductive invariants are found through Algorithm 2, where the function 'rewrite' is given by rewrite system (1).

**Data**: Set of queues $Q$, functions $t_r, t_s$ that produce expressions for enter and exit events for each queue.
**Result**: Set of linear equations $E$

```
for each queue x in Q do
    let enter(x) = rewrite(t_r(x)) ;
    let exit(x) = rewrite(t_s(x)) ;
    Add the equation 'num(x) = enter(x) − exit(x)' to E.
end
```

**Algorithm 2:** Reducing ports

The translation may yield many variables and only as many equations as there are queues. It generates a system of equations that implies every linear inductive invariant that can be expressed in terms of $num(x)$. Key is that we only find single inductive invariants.

**Definition 4.2** (Inductive invariant) We say that a property $p$ is an inductive invariant if $p$ holds in the initial state, and if $p(s)$ implies $p(s')$ if $s'$ is a successor state of $s$.

Note that by our definition, an inductive invariant should not only hold for reachable states, but for all states $s$. Recall that we defined properties and successor states in Section 2.2.1.

This is the statement of the theorem below.

**Theorem 4.1** Given a system of equations that, for each queue $x$, contains the equation $num(x) = enter(x) - exit(x)$ in which $enter(x)$ and $exit(x)$ have been translated. All inductive linear equalities that are expressed solely in terms of $num(x)$ are derivable from that system of equations.

*Proof.* We prove the Theorem by contradiction. Assume there is a linear equality in terms of $num(x)$, which is not implied from the system of equations. Say this equality is $a_0 num(x_0) + a_1 num(x_1) + \cdots = v$.

We argue that the constant $v$ must be zero: Take a sequence of assignments of wires up to time $N$, and repeat it. Given the original invariant up to time $N$, we get a second, namely: $2a_0 num(x_0) + 2a_1 num(x_1) + \cdots = v$. Multiplying the original invariant by 2 gives $v = 2v$, hence $v = 0$.

Replacing the occurrences of $num(x)$ with the definitions of $enter(x)$ and $exit(x)$ yields the equation of primitives: $b_0 \cdot c_0 + b_1 \cdot c_1 + \cdots = 0$ (with $b_i \neq 0$). Here $c_i$ are conjuncts, and possibly $N$ for the empty set. This equation is not implied from the original system of equations, so we know that the left hand side of this equation is not empty. (If this translated to $0 = 0$, the original equality would be implied.) Now we derive a contradiction to $b_i \neq 0$, assuming independent wires.

Let $c_i$ be a smallest conjunct, that is, for no $j$ every $c_j \subset c_i$. Note that the wires in $c_i$ are all fundamental wires, as defined in Definition 2.3. Now let $N = 0$, and assign a value of 1 to all wires in $c_i$ at $t = 0$, while assigning 0 to all other wires. This implies that $c_i = 1$ and $c_j = 0$ for $j \neq i$. Filling in these assignments of $c$ into our equation of primitives gives us: $b_i 1 = 0$, so $b_i = 0$, which is a contradiction. □

The main limitation of this theorem lies in the definition of state, which allows every assignment of fundamental wires to occur. Hardware typically has a state, and even for the queues, the number of packets that can leave typically depends on the history. To give an artificial example, consider two queues that never receive any input, say B1 and B2. From B1, we accept all available packets, while from B2, we do not. Our analysis derives $num(\mathsf{B2}) = 0$, but it will not derive $num(\mathsf{B1}) = 0$ or $num(\mathsf{B1}) = num(\mathsf{B2})$ because it does not recognize that no packets could ever leave B1. In practice, however, such situations are rare to communication fabrics.

Using the previous observations, our algorithm can be summarized as follows:

1. For every queue, the hardware designer gives an expression indicating when a packet enters, and when one leaves.

2. Generate the system of equations with an equation for every queue $x$:

$$num(x) = enter(x) - exit(x)$$

   Translate $enter(x)$ and $exit(x)$ into primitives according to Equations (1)

3. Perform Gaussian elimination to obtain all equalities between $num(x)$.

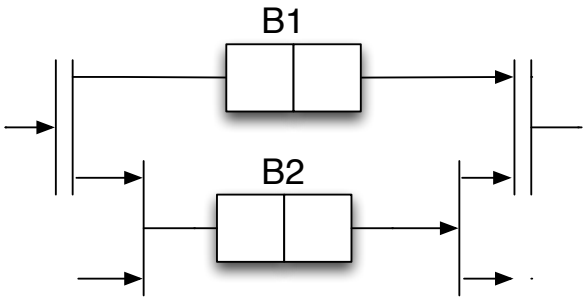



**Figure 4.2:** *A network with message dependency in* B2

### 4.1.6 Data dependent queues

We are also interested in the number of packets of a certain type. To see when this might be useful, consider Figure 4.2. In this figure, queue B2 is preceded by an arbiter, and has a switch at its output. We call the packets that are switched upwards to be of 'type A'. The other packets are of 'type B'. Depending on the particular configuration of this network, B2 might contain as many packets of type A as the number of packets in queue B1. Such an invariant could be necessary to prove progress of this network.

We handle these invariants by adding a different kind of queue:

$$\begin{aligned} num_A(x) &= enter_A(x) - exit_A(x) \\ enter_A(x) &= s(x.\text{in.irdy} \wedge x.\text{in.trdy} \wedge A, N) \\ exit_A(x) &= s(x.\text{out.irdy} \wedge x.\text{out.trdy} \wedge A, N) \end{aligned}$$

Note that the above is only one equation: $enter_A(x)$ and $exit_A(x)$ are only there for readability. It is not necessary to add this equation of every possible type $A$ and every queue $x$. After translating all terms, we inspect every conjunct occurring in this translation. In the case that data wires of the output of B$i$ occur in a conjunct, we add the equation above, with $A$ defined as the conjunction of all the data wires of the output of B$i$. Adding this equation causes a translation of $s$ again, which may introduce new conjuncts. For this reason, we iterate the process until we reach a fixed point.

## 4.2 Step by step analysis

In this section, we go through the network shown in Figure 4.1. The first step is to identify the enter and exit conditions:

$$\begin{aligned}
enter(\mathsf{B1}) &= s(\mathsf{B1.in.trdy} \wedge \mathsf{inR} \wedge \mathsf{B3.in.trdy}, N) \\
enter(\mathsf{B2}) &= s(\mathsf{B2.in.trdy} \wedge \mathsf{B1.out.irdy}, N) \\
enter(\mathsf{B3}) &= s(\mathsf{B3.in.trdy} \wedge \mathsf{inR} \wedge \mathsf{B1.in.trdy}, N) \\
exit(\mathsf{B1}) &= s(\mathsf{B1.out.irdy} \wedge \mathsf{B2.in.trdy}, N) \\
exit(\mathsf{B2}) &= s(\mathsf{B2.out.irdy} \wedge \mathsf{outR} \wedge \mathsf{B3.out.irdy}, N) \\
exit(\mathsf{B3}) &= s(\mathsf{B3.out.irdy} \wedge \mathsf{outR} \wedge \mathsf{B2.out.irdy}, N)
\end{aligned}$$

Note that there are no occurrences of .out.trdy or .in.irdy: these wires are input wires for B1 to B3. Therefore, they can be written in terms of other wires in the circuit. Two of these wires are inR and outR. These stand for whether a packet is ready to be injected by some environment, or whether the environment is ready to receive one.

The next step is to translate each of these conditions. For this circuit, these are fairly trivial, as all conditions consist of conjunctions only:

$$\begin{aligned}
& s(\mathsf{B1.in.trdy} \wedge \mathsf{inR} \wedge \mathsf{B3.in.trdy}, N) \\
=& \{\mathsf{B1.in.trdy}, \mathsf{inR}, \mathsf{B3.in.trdy}\}
\end{aligned}$$

We substitute $num(\mathsf{B1})$ with $enter(\mathsf{B1}) - exit(\mathsf{B1})$ (and do the same for B2 and B3):

$$\begin{aligned}
num(\mathsf{B1}) &= enter(\mathsf{B1}) - exit(\mathsf{B1}) \\
&= \{\mathsf{B1.in.trdy}, \mathsf{B3.in.trdy}, \mathsf{inR}\} - \{\mathsf{B1.out.irdy}, \mathsf{B2.in.trdy}\} \\
num(\mathsf{B2}) &= enter(\mathsf{B2}) - exit(\mathsf{B2}) \\
&= \{\mathsf{B1.out.irdy}, \mathsf{B2.in.trdy}\} - \{\mathsf{B3.out.irdy}, \mathsf{B2.out.irdy}, \mathsf{outR}\} \\
num(\mathsf{B3}) &= enter(\mathsf{B3}) - exit(\mathsf{B3}) \\
&= \{\mathsf{B1.in.trdy}, \mathsf{B3.in.trdy}, \mathsf{inR}\} - \{\mathsf{B3.out.irdy}, \mathsf{B2.out.irdy}, \mathsf{outR}\}
\end{aligned}$$

With Gauss elimination, we can find one equation that does not contain any of the 'set' terms, namely:

$$1 \cdot num(\mathsf{B1}) + 1 \cdot num(\mathsf{B2}) - 1 \cdot num(\mathsf{B3}) = 0$$

By pretty printing negative terms to the right hand side of the equality, while printing positive terms on the left, we get the equality in exactly the desired form:

$$num(\mathsf{B1}) + num(\mathsf{B2}) = num(\mathsf{B3})$$

## 4.3 Conclusions

We presented a fully automatic method to derive inductive invariants from RTL designs of communication fabrics. Our method requires a precise definition of the

events of entering or leaving a queue. From this definition, the remaining analysis is fully automatic. Recent advancements in the verification of communication fabrics all required high-level models. Our approach leverages these advancements to regular RTL designs. This opens up the possibility of studying the scalability of the combination of our invariant generation technique with hardware model-checking techniques for arbitrary RTL designs.

We will see in Chapter 7 that all invariants can be extracted from certain designs with 800 queues in around a second.

# Chapter 5
# Liveness verification

As discussed in Chapter 1, liveness is a class of properties that indicate that an event will eventually trigger a response. In the context of communication fabrics, liveness is both important, and difficult to verify. For this reason, we propose a separate method for it.

We propose a novel approach for liveness verification of RTL designs of communication fabrics. Our approach does not require a high-level model and scales up to designs with hundreds of queues. Similar to all related works, our method is sound but incomplete. As in the previous chapter, we use an abstraction from the details of queue implementations. Queues are replaced with uninterpreted functions and a set of *queue properties*. These properties capture essential characteristics of a queue.

The new idea used here is to express liveness as the average values of wires along infinite runs. Such infinite runs are represented by finite runs with a *lasso* (See Theorem 9, [Biere et al., 1999]). The invariants introduced in the previous chapter are also necessary in our approach. We then describe a network and liveness properties as a Satisfiability Modulo Theories (SMT) problem.

Our SMT encoding distinguishes three behaviours of a wire: the initial value until the start of the lasso, its value at the start of the lasso, and its *average* value over the lasso. Using these average values, liveness of a wire means that its average value is not 0. If a wire is not 0 on average, it is 1 infinitely often. To relate all values, we add several properties to the SMT instance. In doing so, we abstract away from irrelevant timing details. The soundness of our approach follows from the correctness of these individual properties.

## 5.1 Liveness

Given the xMAS protocol introduced in Chapter 2, we define liveness of channel $x$ this way: when $x$.irdy is high, $x$.trdy will eventually be high too. Expressed as an LTL property, a channel $x$ is live if:

$$\Box x.\text{irdy} \Rightarrow \Diamond x.\text{irdy} \wedge x.\text{trdy}$$

If $x$ is the output of a queue as described in Section 2.2.1, we can use that $r_t$ corresponds to $x$.irdy $\wedge$ $x$.trdy, and $r_s$ to $x$.irdy. This leads us to the following

equivalent formula:

$$\Box r_s \Rightarrow \Diamond r_t$$

Given a trace $\sigma_0, \sigma_1, \ldots$, we interpret this LTL formula as follows:

$$\forall i.\ r_s(\sigma_i) \Rightarrow \exists i' \geq i.\ t_s(\sigma_{i'}) \tag{1}$$

Liveness at the input of a queue does not need to be checked, if liveness at its output can be established. Let *in*, *out* be the input and output channel of a queue. Then *in*.trdy will be high if the queue is not full. This happens if *out* is live. Therefore, if *out* is live, then *in* is also live.

Liveness of *out* depends on *out*.trdy, which is determined by other elements. It is even possible for these elements to be outside of the communication fabric. We expect those elements to be modeled as part of the communication fabric design.

## 5.2 A manual proof

We prove liveness of a small example. While doing so, we give general equations that hold in every network. Given these equations, we are able to generalize our technique in Section 5.3.

In this section, we will prove deadlock freedom of a design manually. This design is chosen to be simple, yet require most of the techniques we automated.

### 5.2.1 A simple example

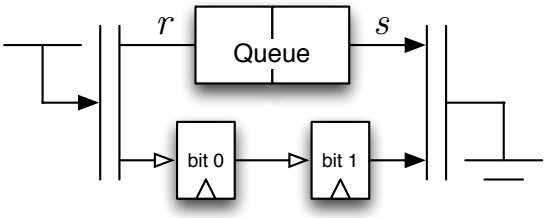



**Figure 5.1:** *A queue and two flops.*

Consider the example in Figure 5.1. It shows a small network consisting of a source injecting packets, two single-bit flops, a queue, and a sink consuming packets. Whenever a packet enters the queue, the first bit is raised. If the first bit is high, and the second is low, the bits swap values at the next clock cycle. A packet leaves the queue if it exists, the sink is available, and the second bit is high. As a packet leaves the system, the second bit is set to zero. Should the first bit be high, the system is considered full and no packets can enter the queue.

The queue has the interface as given in Section 2.2.1. A high $t_r$ puts the packet available on $d_r$ in the queue if and only if the queue is not full.

The components are combined as described earlier. For clarity, we give $t_r$, $t_s$, $r_r$ and $r_s$ for the queue in the system in Figure 5.1, and repeat the next state for the two registers.

We abstract away from the queue implementation, we do not give the entire circuit. However, let $\texttt{bit 0}, \texttt{bit 1} \in F$ be registers in our circuit. As inputs we have $\texttt{sinkReady}, \texttt{sourceReady} \in W$. In order for us to prove liveness, we need to assume fairness of the sink: there will always be a point in time where the sink is ready to accept. In other words, $\sigma_{i'}(\texttt{sinkReady})$ is *true* infinitely often. In general, we assume fairness of all sinks and sources, i.e., the source offers a packet infinitely often, and the sink is ready to accept infinitely often.

The parts of the circuit which we do not abstract away from – the remaining logic – is as follows:

- An available packet is enqueued if the first flop is low, and space is available in the queue.

$$t_r(\sigma) = \neg\sigma(\texttt{bit 0}) \wedge \sigma(\texttt{sourceReady}) \wedge r_r(\sigma)$$

- A non-empty queue can remove its header package if the sink is ready, and the second flop is high.

$$t_s(\sigma) = \sigma(\texttt{bit 1}) \wedge \sigma(\texttt{sinkReady}) \wedge r_s(\sigma)$$

- The first flop keeps a previous 'true' value if it cannot hand over its value to the second. It becomes 'true' after a packet is enqueued.

$$n(\sigma)(\texttt{bit 0}) = (\sigma(\texttt{bit 1}) \wedge \sigma(\texttt{bit 0})) \vee (\neg r_s(\sigma) \wedge \neg\sigma(\texttt{bit 0}))$$

- The second flop keeps a previous 'true' value if no packet is dequeued. It becomes 'true' if it has to take a value from the first flop.

$$n(\sigma)(\texttt{bit 1}) = (\sigma(\texttt{bit 1}) \wedge \neg t_s(\sigma)) \vee (\neg\sigma(\texttt{bit 1}) \wedge \sigma(\texttt{bit 0}))$$

- Initially, the system is empty and the bits are low. Let $\sigma_0$ denote the initial state, then:

$$\texttt{bit}_0(\sigma_0) = \mathit{false}$$
$$\texttt{bit}_1(\sigma_0) = \mathit{false}$$

For this circuit, we wish to treat the queue as a black box, but we still say something about $r_r$ and $r_s$. These depend on the number of packets in the queue: if the queue is not empty, it is ready to send, so $r_s$ is high. If the queue is not full, it is ready to receive, so $r_r$ is high. In order to do so, without having to look at the queue implementation, we define the number of packets in the queue, which is a number that depends on the state. If $\sigma'$ is a successor state of $\sigma$, then:

$$Q(\sigma') = Q(\sigma) + (t_r(\sigma)\,?\,0 : 1) - (t_s(\sigma)\,?\,0 : 1)$$

We are now able to relate the values $r_r$ and $r_s$ to $Q$: let $k \geq 1$ be the size of the queue, then the queue is full if it holds $k$ packets. It is empty if it holds zero.

$$r_r(\sigma) := (Q(\sigma) < k)$$
$$r_s(\sigma) := (Q(\sigma) > 0)$$
$$Q(\sigma_0) := 0$$

### 5.2.2 Liveness proof

We prove liveness of Formula 1 by contradiction, i.e, we are looking for a counter-example. A counter-example would be an infinite run of the system, such that for some $T$:

$$Q(\sigma_T) > 0 \wedge \forall i' \geq T. \neg\, t_s(\sigma_{i'})$$

Using the definition of $t_s(\sigma_{i'})$, we get:

$$Q(\sigma_T) > 0 \wedge \forall i' \geq T. \neg\, \sigma_{i'}(\texttt{bit 1}) \wedge \sigma_{i'}(\texttt{sinkReady}) \wedge r_s(\sigma_{i'}) \tag{2}$$

**Lemma 5.1** (Queue never empty) For $i' \geq T$, $r_s(\sigma_{i'})$.

*Proof.* Induction on $m$ and our assumptions on $Q$ gives us $Q(\sigma_{T+m}) \geq Q(\sigma_T)$, since $\neg\, t_s(i')$ for $i' > T$. From $Q(\sigma_T) > 0$ we obtain $Q(\sigma_{i'}) > 0$, so by the definition of $r_s$, $r_s(\sigma_{i'})$ for $i' \geq T$. □

**Lemma 5.2** (Last flop eventually permanently 0) For all $i' \geq T$, $\neg\sigma_{i'}(\texttt{bit 1})$.

*Proof.* Assume not, and let $i'$ be a value bigger than $T$ for which $\sigma_{i'}(\texttt{bit 1})$. Induction on $m$ gives us $\sigma_{i'+m}(\texttt{bit 1})$ from the definition of $n(\sigma)(\texttt{bit 1})$, using $\neg t_s(\sigma_{i'})$. We conclude that $\sigma_{i'}(\texttt{bit 1})$ for $i' \geq T$.

For some $i' \geq T$ we have $\sigma_{i'}(\texttt{sinkReady})$, by fairness of the sink. Together with Lemma 5.1, this contradicts Equation 2. Therefore, $\neg\sigma_{i'}(\texttt{bit 1})$ for $i' \geq T$. □

**Lemma 5.3** (First flop eventually permanently 0) For all $i' \geq T$, $\neg\sigma'_i(\texttt{bit 0})$.

*Proof.* From Lemma 5.2, $\neg\sigma_{i'}(\texttt{bit 1})$. From the same lemma, $\neg\sigma_{i'+1}(\texttt{bit 1})$, so $\neg n(\sigma_{i'})(\texttt{bit 1})$. Using the definition of $n(\sigma)(\texttt{bit 1})$, it follows that $\neg\sigma_{i'}(\texttt{bit 0})$. □

We have $Q(\sigma_0) = 0$, $\neg\sigma_0(\texttt{bit 0})$ and $\neg\sigma_0(\texttt{bit 1})$. As a packet enters, $Q$ increases, but the flops toggle as well.

**Lemma 5.4** (Invariant) For all $i \geq 0$, $Q(\sigma_i) = (\sigma_i(\texttt{bit 0})\ ?\ 0 : 1) + (\sigma_i(\texttt{bit 1})\ ?\ 0 : 1)$.

*Proof.* By induction on $i$. The base case for $i = 0$ is immediate. Definitions of $Q(\sigma')$, $n(\sigma)(\texttt{bit}_0)$ and $n(\sigma)(\texttt{bit}_1)$ prove the induction step. □

Lemma 5.2 and 5.3 imply $Q(\sigma_T) = (\sigma_i(\texttt{bit 0})\ ?\ 0 : 1) + (\sigma_i(\texttt{bit 1})\ ?\ 0 : 1) = 0$, contradicting Equation 2, which requires $Q(T) > 0$. This proves Equation 1, liveness of our example.

## 5.3 Automated proof

In the liveness proof of the last section, induction is used several times. It is used to prove that a queue is never empty, and that the last flop is permanently 0. These induction proofs are of a similar nature: given that a wire $w$ has a value in state $\sigma$, it will have the same value in successor state $\sigma'$. These are proofs that a wire will permanently hold a value, under certain conditions.

The core idea of our approach is to express the statement 'after time $T$, a wire is permanently zero' as 'the average value of the wire over the lasso is equal to zero'. For 'the average value' to be defined, and for these two statements to be equivalent, we will only look at lasso runs. Lasso runs will be discussed in the next section.

We develop an SMT encoding of the negation of liveness, such as Equation 2, together with relevant information about the circuit. Our encoding is an over approximation of a possible trace of our circuit. This means that all the information added about the trace must be valid, but we do not expect the set of all traces to be defined by the SMT instance. Consequently, the SMT encoding may have solutions, while there is no trace to prove that our circuit is not live.

### 5.3.1 Runs and lassos

We briefly justify the fact that traces of a circuit are represented by finite runs with a lasso. This argument comes from one used to justify bounded model checking (e.g. Theorem 9, [Biere et al., 1999]). Let $\sigma_0, \sigma_1, \ldots$ be a trace in a circuit. Since there are only finitely many possible states of the network, some of the states are in the run infinitely often. Since sinks and sources are fair, the events of offering and accepting packets occur infinitely often as well. This means we can pick two times, say time $T$ and time $T + l$, such that:

- the state $\sigma_T$ is identical to the state $\sigma_{T+l}$
- between time $T$ and time $T + l$, a packet is offered at least once, and sinks are ready to accept at least once

From this, we construct a lasso run $\rho_0, \rho_1, \ldots$ as follows: Up to time $T + l$, our constructed lasso run is identical to the original run, so $\rho_i = \sigma_i$ for $i \leq T + l$. At time $T + l$, all inputs start behaving exactly as they have at time $T$, so $\rho_{i+l} = \rho_i$ for $i \geq T$. Note that $\rho_{i+1}$ is a successor state of $\rho_i$, since $\sigma_{i+1}$ is a successor state of $\sigma_i$. Therefore, our sequence $\rho_0, \rho_1, \ldots$ is a trace. If a channel is not live in the $\sigma$-trace, then there is a $\rho$-trace in which it is not live either. This $\rho$-trace, however, is described completely by its first $T + l$ states. We use this property to define the average value of a wire over an infinite trace.

The value of property $w$ at time $i$ is given by $w(\sigma_i)$. Note that $w(\sigma_i)$ is not a variable that is calculated anywhere in our approach, nor is it one that occurs in the final SMT encoding. We do relate all variables of the SMT encoding to $\sigma_i$, such that any run gives an assignment to the SMT variables. Soundness of our approach will follow from checking every clause added to the SMT instance against its interpretation according to $\sigma$. In the next section, we also write $\sigma(w)$

as part of a clause to indicate an SMT expression in terms of other SMT variables equivalent to $\sigma(w)$.

### 5.3.2 Encoding liveness as averages

To describe the counterexample, we encode the long term behaviour of the property "a packet leaves the queue", i.e. $t_s(\sigma)$. Recall from Section 2.2.2 that such a property is described in terms of fundamental wires $W$.

We define the following variables:

$F_p$ A *persistency variable* for a property $\neg p$. Boolean $F_p$ is true iff: $\neg p(\sigma_i)$ for $i \geq T$.

$a_p$ An *average value* to relate several properties in the lasso, this is a sum in which $p(\sigma_i)$ takes the value 1 if it is high, and 0 if it is low:

$$a_p = \sum_{i=T}^{i+l-1} \frac{p(\sigma_i)}{l}$$

$c_w$ A *current value* for every fundamental wire $w \in W$. This is a Boolean: $c_w = \sigma_T(w)$.

Variables $F_p$ to $c_w$ will be variables in the SMT problem. Their definitions are not encoded in the SMT problem, but used to motivate the properties added to the SMT problem. We can express 'eventually never $p$', by '$F_p$'.

In our example of the previous section, we showed that $r_s(\sigma_T) \wedge F_{t_s}$ leads to a contradiction. Given a queue annotation, we can express $r_s(\sigma_T)$ in terms of $c_w$ values. Therefore, $r_s(\sigma_T) \wedge F_{t_s}$ can be expressed in SMT variables entirely. We let an SMT solver deduce the contradiction for us. To achieve this, we add several clauses that relate these variables.

The first clauses added to the SMT problem directly follow from the definitions of $F$, $a$, $c$ and $c'$.

$$F_p \leftrightarrow (a_p = 0) \tag{3}$$
$$F_w \rightarrow \neg c_w \tag{4}$$
$$F_f \rightarrow \neg \sigma_{T+1}(f) \tag{5}$$

The clauses above are added for all relevant properties $p$ – we will discuss which properties are relevant later, for all fundamental wires $w$, and for those fundamental wires $f$ for which the next value can be expressed in terms of the state $W \rightarrow \mathbf{2}$. In other words: the value referred to in Equation 5 by $\sigma_{T+1}(f)$, can is expressed in terms of the wire driving $f$, thus using the values $c_w$ (for different $w$). Equations 3 to 4 are formulated in SMT format as is, but the expression for $\sigma_{T+1}(f)$ are expanded for Equation 5 based on the circuit.

### 5.3.3 Relating average values

To relate average values, we use that we are in a lasso. This implies that if $\texttt{bit}_1$ is raised, it must also be lowered at some point for the state of the network to return to the current state. Indeed, for register $f$ with driving wire $d$ we know:

$$a_d = a_f \qquad \text{if } d \text{ drives flop } f \tag{6}$$

In addition to relating $F$ to averages, we also relate averages amongst themselves. In particular:

$$a_u \leq a_v \qquad \text{if } u \to v \tag{7}$$

$$a_u + a_v \leq a_w + 1 \qquad \text{if } u \wedge v = w \tag{8}$$

To recognize equalities like $u \to v$ and $u \wedge v = w$, we use the rewrite system from the previous chapter, namely:

$$\begin{aligned} a_{\neg p} &= a_{true} - a_p \\ a_{p \vee q} &= a_p + a_q - a_p \otimes a_q \\ a_{p \wedge q} &= a_p \otimes a_q \end{aligned}$$

Where $\otimes$ is eliminated by using:

$$\begin{aligned} (a_x + a_y) \otimes a_z &= a_x \otimes a_z + a_y \otimes a_z \\ a_z \otimes (a_x + a_y) &= a_x \otimes a_z + a_y \otimes a_z \\ a_x \otimes a_y &= a_{x \wedge y} \end{aligned}$$

If these were averages over a single time unit ($l = 1$), $\otimes$ can be interpreted as multiplication, and the equations would be in the 'Arithmetic sum-of-product format' considered by Minato [Minato and Somenzi, 1997]. To see that for other values for $l$, note that the values for $a$ in their rewritten form are linear equations over real numbers, which are fast in most SMT solvers. Doing the rewriting first (on $v$, using multiplication for $\otimes$), and then taking the average, will result in the same equations as taking the average first, and then rewriting (using $\otimes$ as uninterpreted syntax).

This allows us to write down a linear expression in place of every $a_w$, such that each resulting term $a_c$ has just a conjunction of input bits as $c$. The averages can then more easily be related amongst themselves using Equations 7 and 8.

To add invariants, as in Lemma 5.4, we use the procedure from the previous chapter. We introduce extra integer variables to express the invariants.

- $q_x$ A *state variable* for every queue $x$, which is an integer indicating the number of packets in queue $x$ at time $T$.
- $f_x$ A *state variable* for every flop $x$, which is an integer indicating its output at time $T$.

For flops, this state variable is related to our instance via:

$$c_x \leftrightarrow (f_x = 1) \tag{9}$$

$$\neg c_x \leftrightarrow (f_x = 0) \tag{10}$$

Queues are related to their state as well, which we discuss in relation to other queue properties. In contrast to the linear equalities and inequalities relating average values, these equations are expressed in terms of integer variables.

### 5.3.4 Queue properties.

For every queue of size $size$, we add the following clauses:

$$0 \leq q \leq size \tag{11}$$
$$q = 0 \leftrightarrow \neg r_s(\sigma_T) \tag{12}$$
$$q = size \leftrightarrow \neg r_r(\sigma_T) \tag{13}$$
$$F_{\neg t_r} \rightarrow F_{\neg r_r} \tag{14}$$
$$F_{\neg t_s} \rightarrow F_{\neg r_s} \tag{15}$$
$$F_{t_r} \rightarrow (c_{r_r} \rightarrow F_{\neg r_r}) \tag{16}$$
$$a_{t_r} = a_{t_s} \tag{17}$$

Clause 17 states that for every packet entering the queue in the lasso, it must also leave, for the queue to return to its original state.

Some networks contain data dependencies. Instead of looking at all possible packets in brute force, we form different expressions for several data types. We consider a packet to be of a certain type, if some set of its bits is high. To determine what types to consider, we find out what data bits occur in the averages found among Equation 17. Each average variable $a_{x_1 \wedge x_2 \wedge \ldots}$, containing some data bits in its conjunction (among the $x$'s) constitutes a type of packet. For these types, we add a variable:

$d_x$ state variable indicating the number of packets for a particular data type in queue $x$.

We write the property `dto` to indicate that the output data has the type we care about, and `dti` for the same input type. These clauses are added for each queue, we omit the $x$ to indicate the queue:

$$(d = q) \rightarrow (q = 0 \vee \texttt{dto}(\sigma_T)) \tag{18}$$
$$(d = 0 \wedge c_{r_s}) \rightarrow \neg \texttt{dto}(\sigma_T) \tag{19}$$
$$\neg F_{t_r} \wedge F_{\texttt{dti}} \rightarrow d = 0 \tag{20}$$
$$\neg F_{t_r} \wedge F_{\neg t_r \rightarrow \texttt{dti}} \rightarrow d = q \tag{21}$$
$$F_{t_s \wedge \texttt{dti}} \wedge \texttt{dto}(\sigma_T) \rightarrow F_{\neg r_s \wedge \texttt{dto}} \tag{22}$$
$$a_{(\texttt{enqueue} \wedge \texttt{not\_full} \wedge \texttt{dti})} = a_{(\texttt{dequeue} \wedge \texttt{not\_empty} \wedge \texttt{dto})} \tag{23}$$

### 5.3.5 Summary

In the final SMT instance, we defined the following variables:

$q_x$ Integer indicating the number of packets in queue $x$ at time $T$.

$f_x$ Integer indicating the flop output at time $T$.

$c_x$ Boolean denoting $v(x, T)$.

$d_x$ Integer indicating the number of packets with a certain kind of data in a queue at time $T$.

$F_w$ Boolean $F_{\neg w}$, true iff: $v(w, t) = 0$ for $t \geq T$.

$a_w$ Real: *average value* arising wherever a $F$ is created.

In our implementation, the first three variables can be defined for the entire network. The last four variables are defined where needed: they occur in the property we are trying to disprove, or in a property of one of the other variables. These properties are Equation 3 to 23. Next to these, we add invariants on $q$, $f$ and $d$, and we add the negation of the property that all queues are live.

## 5.4 Conclusions

We presented a novel verification method for liveness properties of RTL designs of communication fabrics. Our method abstracts away from queue implementations and timing details. Liveness properties are expressed using the average values of wires along lasso runs. Properties together with a representation of the network are translated to an SMT instance. It is sound but incomplete.

Experimental results will show that we are able to prove liveness of small and medium sized examples in Chapter 7.

# Chapter 6
# Extraction of xMAS from RTL

Here we give an algorithm that extracts an xMAS high-level model from an RTL design. The input of our flow is a Verilog description together with a specification of which modules represent queues, and when messages enter or leave a queue. After parsing the Verilog, the RTL design can be considered a set of queues and registers interconnected by combinatorial logic. Our approach extracts the micro-architectural structure of the design from this unstructured combinatorial logic. This is achieved by identifying when the transfer of messages depends on synchronisation with other messages or on routing or arbitration decisions. The result is an acyclic xMAS network *transfer equivalent* to the original RTL design. This means that a transfer occurs – a message moves from a queue to another one – in the xMAS model if and only if the same transfer occurs in the RTL design. To ensure that the generated xMAS satisfies the assumptions made by various high-level tools such as deadlock detection tools, we check local properties using a standard model checker.

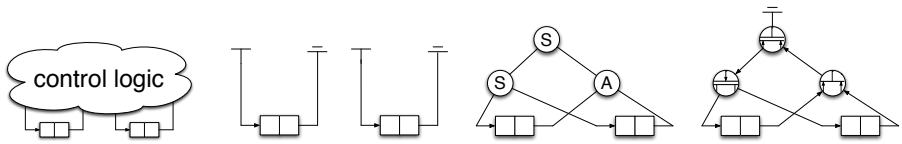
In Section 2.2.1, we defined the notion of port, which is the basic building block for our algorithm. We describe this algorithm in Section 6.1, and its correctness in Section 6.2. There is no definition of what the 'best' abstraction would be, but the resulting abstraction is suitable for verification. Discussion and conclusion follow in Section 6.3 and 6.4

This chapter relies on the definition of port as introduced earlier in Chapter 2.

## 6.1 Translation of RTL to xMAS

A queue gives rise to exactly two ports. For each port the transfer property is known, see Figure 6.1(a). The control logic is modeled as a sink or a source that emits or accepts a packet whenever the routing logic would, see Figure 6.1(b). As a consequence, each such sink or source depends on a large enough part of the original network such that it mimics the original routing logic. This is not a very satisfactory model of a network. The abstraction is sound: the transfer properties are preserved, but it does not tell us much about the communication fabric.

To get a network, we link ports together using xMAS-like elements similar to the merge, join, switch and fork. The xMAS merge and switch are placed into a category we call 'arbiters'. The xMAS join and fork are of the category




(a) Control logic as a jumble of wires (b) Trivial network with identified ports (c) Abstract forest (d) xMAS network

**Figure 6.1:** *Summary of our method*

'synchronizers'. Linking ports together using these two elements for each link, gives rise to a graph in the form of a forest: a set of trees, with an arbiter or synchronizer at each branch, and a port at each leaf. This forest has arbiters and synchronizers at every branch, a single port at the root, and queue inputs and outputs at the leaves, see Figure 6.1(c). Section 6.1.1 gives the translation from identified ports to such a forest. We place an eager source or sink at each root, and replace the arbiters and synchronizers with their respective components, see Figure 6.1(d). Some arbiters become switches, and synchronizers become forks and joins. This is an orientation step, given in Section 6.1.2.

We begin with an informal illustration of how synchronizers and arbiters are created. Our first example has ports $u$ and $v$, and an oracle $P$ modeling packet injection. After parsing the Verilog, assume we obtain the following equations:

$$\begin{aligned} r_u &= B3.\text{trdy} \\ t_u &= \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}, P) \\ r_v &= B1.\text{trdy} \\ t_v &= \texttt{AND}(B1.\text{trdy}, B3.\text{trdy}, P) \end{aligned}$$

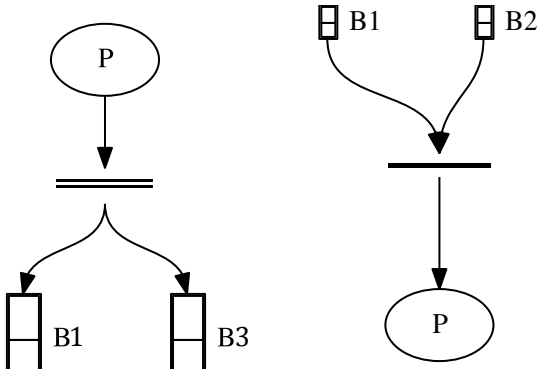
The transfer properties are equivalent in every state, that is, $t_u(\sigma) \leftrightarrow t_v(\sigma)$. We create a synchronizer $s$ with the transfer property of $u$ and as implied property the conjunction of the implied properties of $u$ and $v$. This yields a new port $s$:

$$\begin{aligned} r_s &= \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}) \\ t_s &= \texttt{AND}(B3.\text{trdy}, B1.\text{trdy}, P) \end{aligned}$$

If $u$ and $v$ are both input ports, the synchronizer is refined into a fork. Figure 6.2(a) shows the result. If $u$ and $v$ would be output ports, the synchronizer would be refined into a join.

Our second example (Figure 6.2(b)) has two ports and policy $A$ deciding to which port to give the turn. Packets are accepted when property $P$ holds. After parsing the Verilog, assume we obtain the following equations:

$$\begin{aligned} r_u &= B2.\text{irdy} \\ t_u &= \texttt{AND}(B2.\text{irdy}, \texttt{OR}(A, \texttt{NOT}(B1.\text{irdy})), P) \\ r_v &= B1.\text{irdy} \\ t_v &= \texttt{AND}(B1.\text{irdy}, \texttt{NOT}(\texttt{AND}(A, B2.\text{irdy})), P) \end{aligned}$$




(a) A fork (b) A merge

**Figure 6.2:** *Output of our method*

A SAT solver detects that $t_u \wedge t_v \Rightarrow 0$. We therefore add an arbiter $a$ with $t_a = t_u \vee t_v$.

If $u$ and $v$ are output ports, we refine the arbiter into a merge. If $u$ and $v$ would be input ports, A would be a routing function and we would refine the arbiter into a switch.

### 6.1.1 From ports to a forest

Creating a synchronizer takes two ports and creates a new one. To indicate the new synchronizer, we write a tuple $S\langle x, y\rangle$, where $x$ and $y$ indicate the combined ports: either queue ports, synchronizers or arbiters. When we create an *arbiter*, there is a switching function, or an arbitration policy. A property $t$ will indicate when $x$ should get a turn, containing the information required to implement the switching function or arbitration policy. An arbiter is written as $A\langle x, y, t\rangle$.

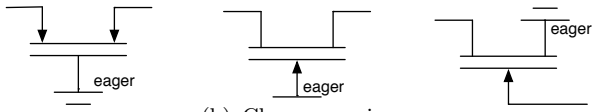
Let $x$ and $y$ be ports. There are two cases in which we combine these ports.

1. If the transfer properties are mutually exclusive: $t_x(\sigma) \wedge t_y(\sigma) \rightarrow 0$ for all states $\sigma$. In this case, we create an arbiter $a = A\langle x, y, t_x\rangle$. We turn $a$ into a port by choosing $r_a(\sigma) = r_x(\sigma) \vee r_y(\sigma)$ and $t_a(\sigma) = t_x(\sigma) \vee t_y(\sigma)$.

2. If the transfer properties are equivalent: $t_x(\sigma) \leftrightarrow t_y(\sigma)$ for all states $\sigma$. In this case, we create a synchronizer $s = S\langle x, y\rangle$. We turn $s$ into a port by choosing $r_s(\sigma) = r_x(\sigma) \wedge r_y(\sigma)$ and $t_s(\sigma) = t_x(\sigma)$.

The reader can verify that the arbiter and a synchronizer, added in the way noted above, preserve $t(\sigma) \rightarrow r(\sigma)$.

To decide whether expressions like $t_x(\sigma) \wedge t_y(\sigma) \rightarrow 0$ are the case, we use a SAT procedure. Using SAT instead of model checking means that we might overlook some cases: we may erroneously decide that $t_x(\sigma) \wedge t_y(\sigma) \rightarrow 0$ does not always hold. In that case we fail to create an arbiter (or a synchronizer in the case of $t_x(\sigma) \leftrightarrow t_y(\sigma)$). It is easy to come up with artificial cases where this happens. In our designs – and in xMAS generated Verilog in general – registers are




(a) Change an output to an input (b) Change an input to an output (c) A circuit equivalent to a wire

**Figure 6.3:** *Different ways of changing the direction of a port*

driven independently. Consequently, a SAT solver gives exactly the same results as a model checker. We did not come across any designs where a model checker provides more insight than a SAT solver, but this is most likely due to the fact that we used xMAS generated Verilog.

Starting out with a set of ports, we combine them using one of the two methods illustrated above. By making as many combinations as possible, we reduce the number of remaining ports as much as possible. For this reason, we give priority to the creation of synchronizers.

After these steps, it may be possible that a port has an implied property which is equivalent to the transfer property. In case the two differ, we add a sink (or a source) that accepts (or provides) a packet in states for which the transfer property is high. This ensures that every port $x$ satisfies $r_x(\sigma) \leftrightarrow t_x(\sigma)$. Such sinks or sources are denoted as $P\langle t \rangle$ for 'port', satisfying $r_{P\langle t \rangle} = t$. In other words: the sink or source is ready to yield a packet at precisely the moment a transfer occurs. We end up with an algorithm for combining ports that gives precedence to adding synchronizers (Algorithm 3, lines 1–17).

After running the algorithm, we obtain a set of xMAS objects with one open end. We discuss what to do with this open end after introducing function orientPortToGraph. The last few lines of Algorithm 3 beginning with 'let $r$', set $r$ and $p$ in a way that do not effect the output of the algorithm. The values for $r$ and $p$ are only used in the correctness proof provided in the next section.

### 6.1.2 Orienting the forest

By connecting the leaves that correspond to the same buffer (one is an input, the other an output), we obtain an undirected graph. To turn this graph into an abstract network, all edges need to get a direction. This turns synchronizers into forks and joins, and arbiters into switches and merges. That step is done by orientPortToGraph: it changes the undirected graph in a directed graph, by orienting all ports.

In this section, we describe how 'orientPortToGraph' may be implemented. We show in the next section that this choice does not matter for the correctness of our translation. For this reason, many orientation algorithms would suffice, and we do not give a pseudo-code algorithm for it.

Independent of the orientation algorithm used, there is always the possibility that one has to deal with non-matching in- and outputs. In other words: the direction of an in- or output will have to be changed. Figure 6.3 shows circuits that achieve this. The first circuit consists of a join connected to an eager sink.

**Data**: Set of ports $C$, maps $t$ and $r$ on $C$
**Result**: Reduced set of ports $C'$

**repeat**
  **for** $x, y \in C$ *with* $x \neq y$ **do**
    **if** $t[x] \Leftrightarrow t[y]$ **then**
      remove $x, y$ from $C$, add $S\langle x, y\rangle$ to $C$;
      let $r[S\langle x, y\rangle] = AND(r[x], r[y])$;
      $t[S\langle x, y\rangle] = t[x]$;
    **end**
  **end**
  **for** $x, y \in C$ *with* $x \neq y$ **do**
    **if** $t[x] \wedge t[y] \Rightarrow 0$ **then**
      remove $x, y$ from $C$, add $A\langle x, y, t_x\rangle$ to $C$;
      let $r[A\langle x, y, t_x\rangle] = OR(r[x], r[y])$;
      let $t[A\langle x, y, t_x\rangle] = OR(t[x], t[y])$;
      break;
    **end**
  **end**
**until** *Nothing changed*;
Create a new set $C'$;
**for** $x \in C$ **do**
  **if** $r[x] \Leftrightarrow p[x]$ **then**
    add orientPortToGraph($x$) to $C'$;
  **else**
    let $r[P\langle t[x]\rangle] = t[x]$; $t[P\langle t[x]\rangle] = t[x]$;
    let $r[S\langle x, P\langle t[x]\rangle\rangle] = t[x]$; $t[S\langle x, P\langle t[x]\rangle\rangle] = t[x]$;
    add orientPortToGraph($S\langle x, P\langle t[x]\rangle\rangle$) to $C'$;
  **end**
**end**

**Algorithm 3:** Reducing ports

This connects to an input, and provides an output with the same transfer property. The second circuit is roughly the same: a fork with an eager source that connects to an output, providing an input.

Since so many implementations suffice for orientPortToGraph, we could ask what would be a 'best orientation'. Two considerations can be made here: one is to keep the resulting circuit as small as possible. This means that we wish to minimize the number of direction changes, as this introduces two components (shown in Figure 6.3). Another consideration we could make is that switches depend on data, while arbiters do not. If we know which wires are responsible for data, we can base our choice on the transfer properties of an arbiter.

We investigate how to minimize the number of direction changes. We can assume that the direction of ports is fixed. To indicate the direction, we will call input ports 'down', and output ports 'up'. This corresponds to the direction of a tree that has the root on top, and input- and output- sides of queues as leaves at the bottom.

A synchronizer with three ports will treat all of them the same: they have a transfer simultaneously. For the three connections leading to a synchronizer, at least one must be an input, and at least one an output. Depending on the third, we then create a fork or a join. This means that when we encounter a synchronizer, we can first orient the underlying branches. If both branches are up, or both are down, the synchronizer is fixed (up or down respectively). Otherwise, we can choose to orient the synchronizer freely.

Since a synchronizer can create a 'freely orientable' node, this means that there are three cases to consider:

1. A node is oriented 'down', which means that requiring it to be oriented 'up' will introduce one of the circuits of Figure 6.3, costing exactly two components.

2. A node is oriented 'up': requiring it to be oriented 'up' introduces one of those circuits.

3. A node is oriented 'free': the underlying circuit size is independent on how the node is oriented.

For the synchronizer, if at least one of its branches is freely orientable, we can always orient it such that one of its branches is an input, and the other is an output. In such a case, the synchronizer node is freely orientable as well.

An arbiter has stronger requirements: all of its branches must be 'down', in which case it is a switch, or all of its branches must be 'up', making it a merge. If we try to give an orientation with the least number of direction changes, we can apply the following strategy: count the number of branches that are 'down', and the number of branches that are 'up'. If one of them is a majority, choose that orientation for all branches. This possibly re-orients some of its branches. If neither of them is a majority, this node becomes 'free', and the way we orient our branches depends on how the parent node would like to orient this node.

A final consideration is what to do at the top node, which does not have a parent. If the top node is an arbiter, we can only choose to orient it and add a source or a sink at the top. In the case of a synchronizer that is oriented 'up' or 'down', we do the same. A free synchronizer with two branches, however, can be omitted altogether: it is necessarily possible to orient the two branches as 'up' and 'down' or vice versa. These two channels 'fit': they can simply be connected.

## 6.2 Resulting Graph Correctness

We show that the resulting graph is equivalent to the network in the original circuit, using the semantics for the xMAS components as presented in Chapter 2.

On the RTL design, we require the different input and output ports together with the corresponding property $t$ to indicate that a transfer occurs, to be known. At the xMAS level, we know all channels and their transfer conditions. We say that an xMAS model and an RTL design are *transfer equivalent* if and only if their transfer conditions are equivalent. *Transfer equivalence* is defined as follows:

**Definition 6.1** (Transfer equivalence) Let $X$ be a set of queue inputs and outputs, and let $f_1$ and $f_2$ be functions that give the transfer property for every element of $X$. We say that $f_1$ and $f_2$ are transfer equivalent if for all elements $x \in X$ and all states $\sigma$: $f_1(x)(\sigma) = f_2(x)(\sigma)$.

**Lemma 6.1** Let $C$ be the inputs and outputs of the queues in a circuit with the transfer properties $t_x$ for each $c \in C$. Let $t'_c = c.\text{irdy} \wedge c.\text{trdy}$ where $c.\text{irdy}$ and $c.\text{trdy}$ follow the xMAS semantics as extracted from $C$ by the proposed algorithm. Then $t$ and $t'$ are transfer equivalent.

*Proof.* To show equivalence, we just need to show that $t \leftrightarrow o.\text{irdy} \wedge o.\text{trdy}$ on a queue output $o$, and that $t \leftrightarrow i.\text{irdy} \wedge i.\text{trdy}$ on a queue input $i$. We show that the following properties hold for all ports inductively:

Each input corresponding to port $c$:

$$t_c \rightarrow c.\text{trdy} \quad (1)$$
$$c.\text{trdy} \rightarrow r_c \quad (2)$$
$$\neg t_c \wedge r_c \rightarrow \neg c.\text{irdy} \quad (3)$$
$$t_c \rightarrow c.\text{irdy} \quad (4)$$

Each output corresponding to port $c$:

$$t_c \rightarrow c.\text{irdy} \quad (5)$$
$$c.\text{irdy} \rightarrow r_c \quad (6)$$
$$\neg t_c \wedge r_c \rightarrow \neg c.\text{trdy} \quad (7)$$
$$t_c \rightarrow c.\text{trdy} \quad (8)$$

Taken together, these equations imply $t_c \leftrightarrow c.\text{irdy} \wedge c.\text{trdy}$.

The algorithm combines ports, building a tree structure. On each port that remains, the underlying tree is connected to an eager sink or source. We perform induction over each such tree structure in two directions. We first do induction over the top equations (queues to trunk), and then - using these equations - over the bottom equations (trunk to queues). Both proofs are by a case analysis over the components. We start with the base cases.

For the queue output $c$ corresponding to port $x$, we know $r_x = c.\text{irdy}$ (Equation 6 holds), and for the queue input $c$, we know $r_x = c.\text{trdy}$ (Equation 2 holds). By $t_x \rightarrow r_x$, Equations 1 and 5 hold. At the 'trunk' side of the network, $c \in C'$, there is an eager sink or source. If $c$ is an input then $c.\text{irdy}$ is high (eager sink), and $c.\text{trdy}$ is high if $c$ is an output, thus Equations 4 and 8 hold. For the corresponding port $x$ we know $r_x \leftrightarrow t_x$, so Equations 3 and 7 hold.

The induction step is by case distinction on the xMAS component and the port direction. The proof for the bottom two equations (Equation 3, 4, 7 and 8) depends on that for the top two (Equation 1, 2, 5 and 6), but not vice versa.

Suppose the port $o$ is an output, and the xMAS component is a join, originating from a synchronizer with its inputs connected at the outputs at $a$ and $b$. By construction of the synchronizer we have $t_o = t_a = t_b$. We know that $t_a \rightarrow a.\text{irdy}$ and $t_b \rightarrow b.\text{irdy}$ by induction for Equation 5. Together with $o.\text{irdy} := a.\text{irdy} \wedge b.\text{irdy}$ for the xMAS semantics of the join, this yields $t_o \rightarrow o.\text{irdy}$, proving the induction step for Equation 5. We can do the same for Equation 6, yielding $o.\text{irdy} \rightarrow r_o$ by $a.\text{irdy} \rightarrow r_a$ and $b.\text{irdy} \rightarrow r_a$ and the synchronizer property $r_o = r_a \wedge r_b$.

We now perform the induction step in the other direction. We are still using that $t_b \rightarrow b.\text{irdy}$, so $t_a \rightarrow b.\text{irdy}$. By induction $t_o \rightarrow o.\text{trdy}$, so $t_a \rightarrow o.\text{trdy}$. The xMAS definition of a join $a.\text{trdy} := o.\text{trdy} \wedge b.\text{irdy}$, hence $t_a \rightarrow a.\text{trdy}$, Equation 8. Similarly, $\neg t_o \wedge r_o \rightarrow \neg o.\text{trdy}$. Using the properties of a synchronizer, we get $\neg t_a \wedge r_a \wedge r_b \rightarrow \neg o.\text{trdy}$. Using the xMAS definition of a join: $\neg o.\text{trdy} \rightarrow \neg a.\text{trdy}$,

so $\neg t_a \wedge r_a \rightarrow \neg a.\text{trdy}$, Equation 7. For $b$ we can supply the same proofs by symmetry.

The case we treat is where $o$ is an output, the xMAS component is a merge, originating from an arbiter with its inputs connected at the inputs at $a$ and $b$, and arbitration policy $t_a$. Inductively, $t_a \rightarrow a.\text{irdy}$ and $t_b \rightarrow b.\text{irdy}$, so $t_a \vee t_b \rightarrow a.\text{irdy} \vee b.\text{irdy}$, which yields Equation 5 for $o$ through the definition of the merge. The same reasoning applies to Equation 6, so we focus on the induction step in the trunk to queue direction. The definition of the merge yields $a.\text{trdy} := o.\text{trdy} \wedge t_a \wedge a.\text{irdy}$. We have $t_a \rightarrow a.\text{irdy}$ from the previous induction, $t_o \rightarrow o.\text{trdy}$ from this one, yielding $t_a \rightarrow o.\text{trdy}$ since $t_o = t_a \vee t_b$. Combining these observations yields $t_a \rightarrow a.\text{trdy}$ (Equation 5). The other port has $b.\text{trdy} := o.\text{trdy} \wedge \neg t_a \wedge b.\text{irdy}$. The arbiter is only created when $t_b \rightarrow \neg t_a$ (by $t_b \wedge t_a \rightarrow 0$), so Equation 5 is satisfied using the same observations as for $a$. Equation 7 holds for port $a$ immediately by definition of $a.\text{trdy}$. For port $b$, we use the way an arbiter was created to conclude $\neg t_b \rightarrow t_a \vee \neg t_o$ and $r_b \rightarrow r_o$. Together, this yields $\neg t_b \wedge r_b \rightarrow (\neg t_o \wedge r_o) \vee t_a$. By induction $\neg t_o \wedge r_o \rightarrow \neg o.\text{trdy}$, from which we may conclude $\neg t_b \wedge r_b \rightarrow \neg b.\text{trdy}$ by the xMAS definition of $b.\text{trdy}$. This completes the induction step for a merge.

The other output ports (networks used for reversing direction, and sources) are left to the reader. For the input ports, the proofs are roughly symmetric and left for the reader as well, with the exception of the switch (the most involved component). Let $i$ be an input, and the xMAS components be a switch, originating from an arbiter with its outputs connected at the inputs $a$ and $b$, and arbitration policy $t_a$.

By induction: $t_a \rightarrow a.\text{trdy}$ and $t_b \rightarrow b.\text{trdy}$, so $t_a \vee t_b \rightarrow (a.\text{trdy} \wedge t_a) \vee (b.\text{trdy} \wedge \neg t_a)$. The switch is defined as $i.\text{trdy} := (a.\text{trdy} \wedge t_a) \vee (b.\text{trdy} \wedge \neg t_a)$, while the arbiter has $t_o = t_a \vee t_b$, resulting in Equation 1 for $i$. Similarly: $i.\text{trdy} \rightarrow a.\text{trdy} \vee b.\text{trdy} \rightarrow r_a \vee r_b$, resulting in Equation 2 for $i$. The induction step in the other direction is symmetric to that of the merge and left to the reader.

This completes our induction, which implies that $t_c \leftrightarrow c.\text{irdy} \wedge c.\text{trdy}$ for all queue in- and output ports. □

The proof is independent of choices from the orientation function, so orientation of synchronizers and arbiters does not change whether transfer equivalence holds. In the networks we obtain, we have the liberty to change merges into switches, forks into joins, and vice versa.

## 6.3 Discussion

Our method also applied to virtual channel examples, which were analyzed in related works [Chatterjee et al., 2012; Ray and Brayton, 2012]. Similar to these related works, our experiments are performed on xMAS generated code. One bias is that these models already contain architectural insight. This bias appears in the generation of the Verilog which is done by translating each xMAS module separately. Note that the design is fully flattened before being processed by our method. The latter captures the parts of a design where packets synchronize with each other or where the progress of packets depends on arbitration or switching decisions. The key element is to identify when transfer properties are mutually

exclusive or equivalent. In more realistic and larger cases, these properties are larger and more complex. Still, they always involve a relatively small subset of the design and therefore should stay within the capabilities of SAT solvers.

Additionally, all registers in our designs are independent. As a result, using SAT solvers instead of model checkers suffices. When a single wire is the driving wire for two different registers, this is not the case. We can craft a network in which a synchronizer is not detected automatically, because two queues use two different registers that always have the same value. There are industrial examples in which this situation arises. The straightforward solution is to use model checking instead of SAT solving. Another solution is to pre-analyze the network in the spirit of Chapter 4, and identify such registers.

A final bias is that we only produce xMAS networks in which every cycle contains a buffer. In other words: the method fails to recognize structure when applied to RTL circuits with a cycle without a buffer.

In all cases where our method fails to detect structure, it returns a non-eager source or a sink at that place. The translation to a model checking problem can help to distinguish between output which is desired, and output in which still some structure is missing. In other words: it is feasible to detect *all* places where structure is missing. Future work should be able to fill all these gaps, by further investigating the cases where they arise.

Thanks to observations at the xMAS level, intended queues can be implemented more efficiently. The trouble is that once we use these observations, the implementation is no longer pure xMAS. Our approach obtains the routing logic between queues, in a way tolerant for several different queue and buffer implementations.

## 6.4 Conclusion

We presented an algorithm that automatically extracts the micro-architectural structure of RTL descriptions of communication fabrics. We proved that the original RTL circuit is *transfer* equivalent to the extracted micro-architecture. Our approach will be applied on typical examples found in the literature in the next chapter.

Bridging an important gap between unstructured RTL designs of communication fabrics and their micro-architectural structure, our approach provides designers with key insight into their design. More importantly, micro-architectural models play a key role for verification. Until now, the manual construction of an xMAS model was required to apply all these techniques. Our approach removes this barrier. This opens up new ways to use previous works and tackle the challenge of directly verifying RTL designs of communication fabrics.

# Chapter 7
# Experimental results

The previous chapters describe methods to analyse communication fabrics. This chapter tests those methods on a set of designs.

We describe a set of designs in Section 7.1. These designs fall into two categories: designs used to clearly illustrate our approach, and designs taken from previous publication about xMAS, to show that the approaches work on such designs as well. In both cases, we use the xMAS language to describe communication fabrics, and build the corresponding Verilog code.

The queue interfaces to our designs are annotated as described in Section 2.2.2. The same annotations are used for all our algorithms. We clustered most of the algorithms done for this thesis in a new tool called Voi. The examples used in this section can be found at the same link as the tool: `http://sjcjoosten.nl/voi/`

## 7.1 Investigated designs

In Chapter 2, we introduced some xMAS primitives. Although we see queues on an abstract level, other tools do not. It is only for this reason that the implementation of the queue matters. Our techniques give the same results with any queue of size 2 or larger – queues of size 1 do not have a state in which reads and writes can occur at the same time. Similarly, most techniques do not suffer from having large packets, that is: data lines consisting of many bits. A comparison involving very large queues, say of depth 256, with 64-bit packets, may force timeouts in the analysis of other tools, while our methods would still work. A comparison with very small queues, say of size 2, may not show the benefits of our methods.

### 7.1.1 Virtual channels with buffer

Figure 7.1 shows a design proposed by Sayak Ray and Robert Brayton [Ray and Brayton, 2012]. Packets in the queue labeled as 'Buffer' are identified by a bit, such that packets originating from In1 are routed to B3, and those originating from In2 are routed to B4. The arbiter alternates between accepting packets from In1 and In2 if one is offered. The arbiter is persistent in the sense that if the

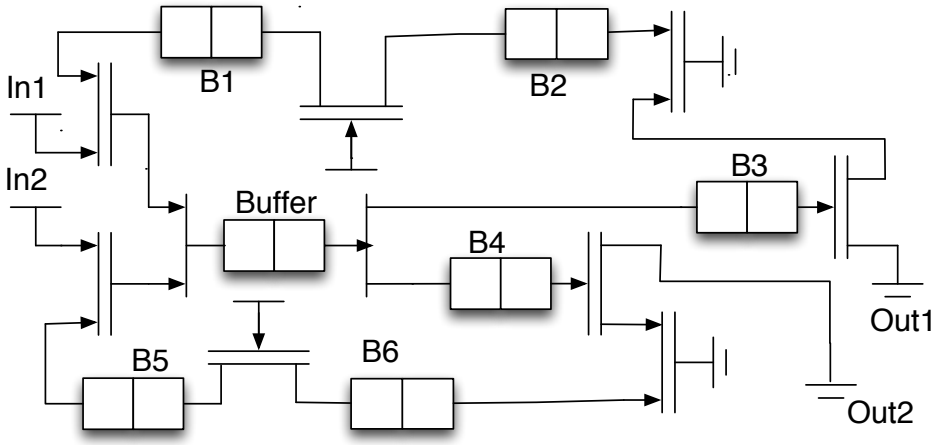



**Figure 7.1:** *Virtual channels and a message dependent buffer*

'Buffer' is full, and the packet from In1 (or In2) is offered, at the next clock tick the packet from In1 (In2) is offered to the Buffer again.

We can vary the queue sizes: the size of queue B2 limits the number of packets in B1, Buffer and B3. If the sizes of B2 and B6 together do not exceed the size of Buffer, a blocked channel from In1 to Out1 will not cause a blocked channel from In2 to Out2. In this sense, the available positions in B2 and B6 model credits for the rest of the network, which are stored in B1 and B5 respectively, and returned when the packets leave B3 and B4.

### 7.1.2 Two-entry scoreboard

A two-entry scoreboard is shown in Figure 7.2. The general idea behind the scoreboard is that transactions – modeled as xMAS packets – come in from the left, obtain a 'tag', and are processed in two phases. The two phases are modeled through the channels at the bottom of the figure: each phase has an outgoing and a returning channel. After processing, transactions are retired on the right. The scoreboard contains the logic to track transactions of two types.

This example originates from a paper intended to illustrate the expressive power of xMAS, by Satrajit Chatterjee et al. [Chatterjee and Kishinevsky, 2012]. The same example is later seen in a paper by Sayak Ray [Ray and Brayton, 2012]. The Verilog implementation used in both of these works was not available to us, so we reimplemented the designs based on the figures in those papers.

The size of B3 and B4 can be chosen such that at most one transaction of each type is allowed into the scoreboard simultaneously. The switch on the left, connecting to B5 and B10, is such that packets with bit `[3]` set are routed to B5, while packets that do not have that bit set are routed to B10. The packets in queues B1 to B4 model tokens.

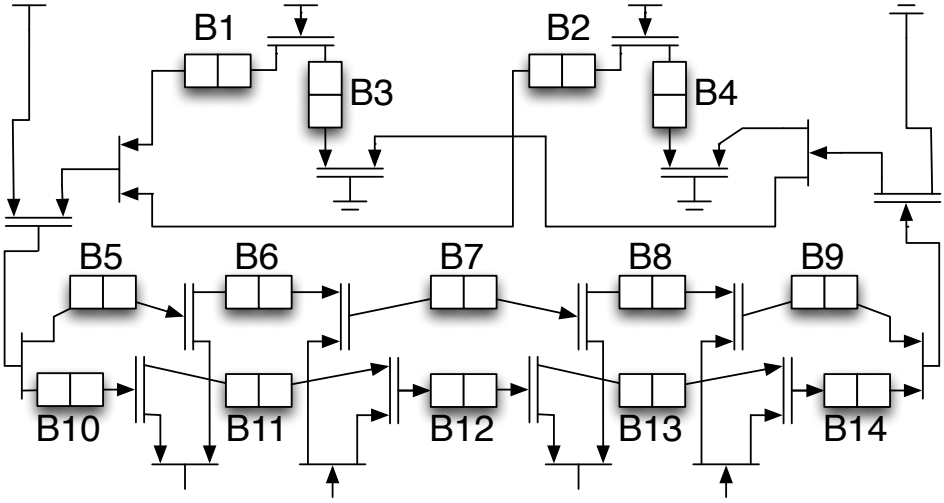
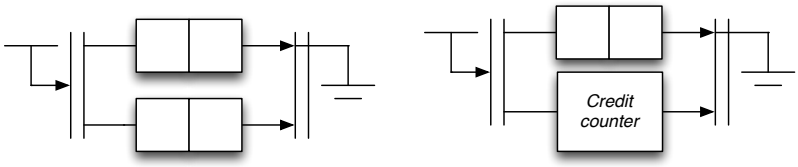



**Figure 7.2:** *A 2-entry scoreboard*

### 7.1.3 Parallel queues




(a) Two queues. (b) A credit counter with a queue.

**Figure 7.3:** *Two layouts with parallel queues. On a more abstract level, the designs are the same. In their implementation, the two designs might differ, although they exhibit the same behavior.*

Figure 7.3 shows how a queue can be placed in parallel with something else: packets accepted from the source are put in the top queue, and something is added to the bottom component as well. In both designs, queues are initially empty, and so is the credit counter. On an abstract level, this makes the two layouts the same. The idea behind these two designs is that it allows us to investigate to what extent state holding elements need to be abstracted.

We implement the credit counter in Figure 7.3(b) using registers, but only annotate these registers, as described in Section 2.2.2. If the size of the credit counter is $2^n - 1$ for some $n$, all of these register states will be reachable. Designs where the size of the credit counter is not $2^n - 1$ will have unreachable states. This property allows us to clearly distinguish between designs which we can, and designs which we cannot fully analyse.

## 7.2 Invariants

A queue can be identified using the properties `inputTransfer`, `inputData`, `outputTransfer` and `outputData`. The analysis of invariants takes a look at those annotations starting with `annotate queue`, and finds invariants in the manner explained here. For this analysis, `outputData` can contain output wires of the queues, but not expressions made from those wires. From the network described in the original chapter, Figure 4.1, we obtained exactly the desired invariant.

Speed measurements in this section are performed on one 1.7 GHz I7-4650U core[1]

### 7.2.1 Virtual channels with buffer

We tested our approach on various credit-flow systems by Ray and Brayton. Figure 7.1 shows the configuration of a buffered virtual channel. The buffer labeled as 'Buffer' can contain two types of packets: those for B3 and those for B6. The last bit of the packet data is used to indicate this.

Running our tool results in the following output:

```
B0 + buffer[0] = B1 + B3 + buffer
B4 + B5 + buffer[0] = B6
```

The variable `buffer[0]` indicates the number of packets in which bit 0 (the last bit) was set. In the first invariant, `buffer - buffer[0]` can be read as: packets in which bit 0 was not set. These two invariants are exactly the invariants required in the analysis by Ray and Brayton [Ray and Brayton, 2012]. Generating these invariants took 0.06 seconds for our implementation.

### 7.2.2 Two-entry scoreboard

In this example, the top right switch causes B9 and B14 to have a data dependency. Because of this, all buffers B5 to B14 have data dependencies. Analysing the network took 0.11 seconds. The invariants found were slightly different from the invariant the authors added:

```
B10[0] + B11[0] + B12[0] + B13[0] + B14[0] + B4 + B5[0] +
B6[0] + B7[0] + B8[0] + B9[0] = B10 + B11 + B12 + B13 + B14
    + B2 + B5 + B6 + B7 + B8 + B9

B3 = B1 + B10[0] + B11[0] + B12[0] + B13[0] + B14[0] +
    B5[0] + B6[0] + B7[0] + B8[0] + B9[0]
```

Note that for our network, B10 does not receive any packets with bit `[0]` set. For this reason, we can deduce `B10_[0] = 0`. Applying this argument iteratively, we obtain `B11_[0]=B12_[0]=B13_[0]=B14_[0]=0`. On the other side, we know: `B5_[0]=B5`, and apply this argument to the next buffers. These two arguments suffice to get the same two invariants as those of Ray and Brayton, namely:

[1] Using one core increases the base frequency to 3.3GHz.

```
B10 + B11 + B12 + B13 + B14 + B2 = B4
B1 + B9 + B8 + B7 + B6 + B5 = B3
```

The above invariant would become invalid if we simulate the network from an (unreachable) state in which `B10_[0]=B10=1`. This implies that the invariant (which the authors of the paper found by 'manual inspection') is not one step inductive by itself for our implementation. The authors mentioned that their one step induction technique failed to prove the invariant for this scoreboard.

An alternative way to obtain the invariant, is to change the network such that it holds. Do to so, change Buffers 5 to 14 such that they are three bits wide instead of four, so no distinction can be made on the last bit `[0]`.

### 7.2.3 Parallel queues

In the case of Figure 7.3(a), a linear invariant is generated:

$$q_{top} = q_{bottom}$$

For our implementations in Figure 7.3(b), the credit counter is not abstracted as a queue. Instead, the internal state is given by the state of some flops in the credit counter. We consider two implementations for a credit counter of size 3. In one implementation, we count the number of credits in a unary way, similar to a shift register. Our method finds the following invariant:

$$q_{top} = f_0 + f_1 + f_2$$

In a different implementation, we use only two flops, and count binary: packets may enter if either bit is zero. Once again, our method finds the desired invariant:

$$q_{top} = f_0 + 2 \cdot f_1$$

**Limitation of invariants** If we make a slight modification, and decide that the credit counter should hold at most two credits, we can get an invariant for the unary encoding:

$$q_{top} = f_0 + f_1$$

For the binary encoding, however, we cannot find any invariants. This occurs because the unreachable state $f_0 = f_1 = 1$, which corresponds to three packets, does not satisfy the invariant one-step inductively. If we allow a packet to enter (or leave) in such a case, the invariant does not hold for that transition. This example shows that it can be beneficial to use the same abstraction for counters as for queues.

### 7.2.4 Scalability of the approach

To test the performance of our method, we repeated the configuration in Figure 7.1 several times by connecting the outputs to the inputs of another design. The results are shown in Figure 7.4. In this case, each repetition of the virtual channels

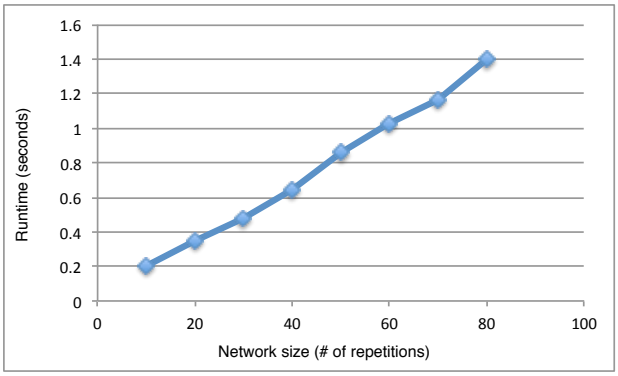



**Figure 7.4:** *Finding invariants. Runtime for repetitions of Figure 7.1*

contains 11 buffers, so the 80 repetition example has 880 buffers. It generated 160 invariants. Also, the invariants were generated as two local invariants per repetition (as we would expect), and not as a linear combination of those invariants. Note that the invariant analysis does not use the fact that the same network is repeated several times.

## 7.3 Deadlock verification

We did not combine the deadlock verification analysis into the tool Voi[2]. Instead, we use a parser written in ACL2 to analyse the Verilog. The resulting structure needs to be copied into a separate file, which we call EMOD module. This makes the deadlock verification more cumbersome to use, but also provides a first step towards obtaining formally verified deadlock-free networks in a theorem prover.

### 7.3.1 Verification Flow

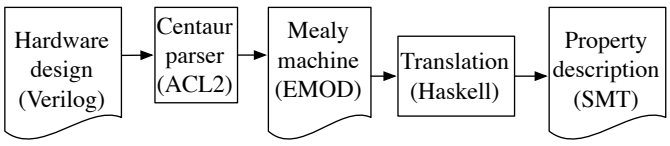



**Figure 7.5:** *Verification Flow for the EMOD framework.*

Figure 7.5 shows the different steps of this approach. The input to our method is an RTL description of the on-chip network architecture using the Verilog hardware description language. The Verilog file is parsed and interpreted using a parser developed by Centaur Technology [Hunt and Swords, 2009]. These files are translated to one EMOD module. During this translation, queues are identified by

[2]When the analysis is integrated, the Voi website will reflect this.

special tags, and their inner working is hidden as a black box. The rest of the design is identified as combinatorial logic and flops. An SMT instance is created from our definition of liveness, and from properties that hold for the combinatorial logic, flops, and queues.

The Centaur translator targets a subset of Verilog. Arithmetic and bitwise operations are supported, as well as non-blocking assignments. A (syntactic) limitation is that no blocks can be used. Transistor-level constructs, real variables, hierarchical identifiers, and multi-dimensional arrays are not supported. After this translation, all wires are assigned Boolean expressions, in which `X` and `Z` values can be regarded as input wires.

We recognise which wires are used for queues, and abstract away from the queues themselves. After a tree of modules is built, each queue in the list of those identified by the hardware designer is syntactically replaced by a function. Flops are treated similarly. Concretely, the EMOD file expresses values using AND, XOR and NOT, but also using queue specific functions. To give an example, the following value could determine whether a transfer occurs in a sequence of two queues:

```
(AND (not_empty Q1) (not_full Q2))
```

If the original RTL contained any cycles, the translation to EMOD gets rid of them, or fails.

In the following sections, we give some artificially constructed examples to indicate when our approach works, and when it does not.

### 7.3.2 Parallel queues

Figure 7.3 from the previous section shows an example for which our approach works, as well as an example for which it does not. Like the queue, the credit counter accepts tokens until it is 'full'. While full, it prevents packets from entering at the source. Similarly, the top queue is prevented from releasing its packets in case the credit counter is empty. This means that in the unreachable case that the top queue is full, and the bottom half is empty, the system is in deadlock.

These designs illustrate that the invariants added to the SMT instance prevent us from getting false positives. Indeed: cases for which an invariant is found are proven deadlock-free, whereas the cases where no invariant is found yield false deadlocks.

### 7.3.3 Buffered virtual channels

We again direct our attention to the example in Figure 7.1. Using our method, the data dependent invariants corresponding to this network are found, and every buffer in the network is proven live (provided all sources and sinks are fair).

**Limitations of one-step simulation** We can modify the arbiters' behaviour such that it allows packets from In1 more often than the packets from In2. For instance, it can take two packets from In1, and then only one from In2. In this case, the properties presented so far are insufficient to verify that B5 is live (or

that packets from In2 are eventually accepted). To prove this, one could add these equations to the SMT instance:

$$T_w \to v(w, n+2) \quad F_w \to \neg v(w, n+2)$$

In essence, these equations add a one-step simulation to our analysis. The equations above allow us to perform a two-step simulation. Unfortunately, adding these equations has a severe impact on the performance of the analysis.

### 7.3.4 Other networks and scalability

Next to the networks described above, we have analysed the network in Figure 7.1 without the queue called Buffer (with its outputs and inputs connected with a wire instead). In all cases, we proved liveness of all queues, with extra queues added at the sources and sinks (thereby verifying their liveness as well), and under the assumption of fairness of sources and sinks.

To illustrate the scalability of our approach, we took the network in Figure 7.1. As with the invariant generation, the network is cloned several times, connecting the outputs to the inputs (Out1 to In1, Out2 to In2). All queues could be verified to be live. The time it took to verify this property is shown in Figure 7.6.

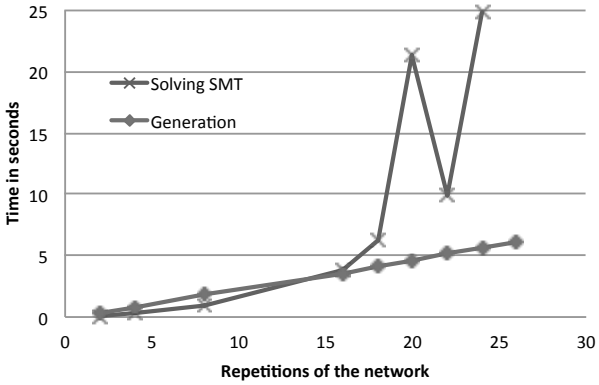



**Figure 7.6:** *Execution times for cascaded buffered virtual channels.*

The measurements in this section were performed using one core on a 1.8 GHz Intel Core i7-2677M[3], using Z3 as the SMT solver. A network with 25 repetitions containing 279 queues (25 repetitions of 11 queues - for of which are at sinks or sources, plus four queues for the unattached sinks and sources) is analysed in 25 seconds. When investigating a network with 26 repetitions, we aborted the execution after 10 minutes. We consider 280 queues to be a rough limit for networks with medium complexity, at which the current SMT solvers tend to fail for solving the instances.

[3]Using one core increases the base frequency to 2.9GHz.

## 7.4 Extracting xMAS from RTL

The extraction of xMAS has been fully implemented in Voi. The switch `-xgraph` generates a dot file for graphviz. The switch `-forest` generates an ASCII representation of the forest before orienting it. The forest can be generated without looking at the 'ready' properties, so `-forest` does not take those into account. CPU speed measurements on this section are performed on four 1.7 GHz I7-4650U core processors. The SAT solver uses all four cores, the rest of the algorithm is sequential.

We used a Haskell interface to minisat built by Niklas Eén [Eén and Sörensson, 2004]. This SAT solver takes conjunctive normal forms as input (instead of Boolean formulas), which requires an additional translation step. The solver runs in Haskell's IO-monad and allows incremental solving. This allows us to translate the entire circuit to a single SAT instance, and then query it for its properties.

In our designs, we could automatically rewrite 4-valued logic to 2-valued logic. In interconnects that use busses, this may not be the case. Our tool Voi handles 4-valued logic, but this feature has not been thoroughly tested, since the considered designs do not rely on it.

Figure 7.2 shows a network of a two entry scoreboard with two phases. The top half models tokens preventing an overflow of packets entering the bottom half. The switch at the bottom left is configured such that tokens from B1 are used to route data to B5, while tokens from B2 are used in packets for B10. To illustrate our approach, we applied our method to Verilog generated from this xMAS model. Our method only uses a Verilog description in the form of the ports of each queue module. The output of our method is a graph visualised using the graphvis program 'dot' [Ellson et al., 2002]. To generate the graph, the call to Voi becomes:

```
time voi xmas_annotations scoreboard.v -xgraph -m mod | dot
          -Tpdf -oscoreboard.pdf
```

This generation took 0.8 seconds, of which 0.15 seconds are due to Voi.

Figure 7.7 shows the structure extracted by our method from the scoreboard example Figure 7.2. Properties used at sources and sinks are labeled with P and a number. While our method largely reconstructed the original design, a few subtle differences arise. For instance, B2 is connected to B10 in the extracted model, while B2 is connected to B5 and B10 in the original xMAS model. Because the switch is configured properly, tokens from B2 are actually routed to B10. Our extraction method makes this fact visible. Also, our method focuses on the control logic. The 'ready' signal of data coming from the input in the top left corner is hidden in P1.

Figure 7.8 shows the result for buffered virtual channels. Its generation took 0.76 seconds, of which 0.08 seconds are due to Voi.

### 7.4.1 Scalability

To test scalability, we again use several copies of the buffered virtual channels. The number of calls to the SAT solver is quadratic in the number of ports. The

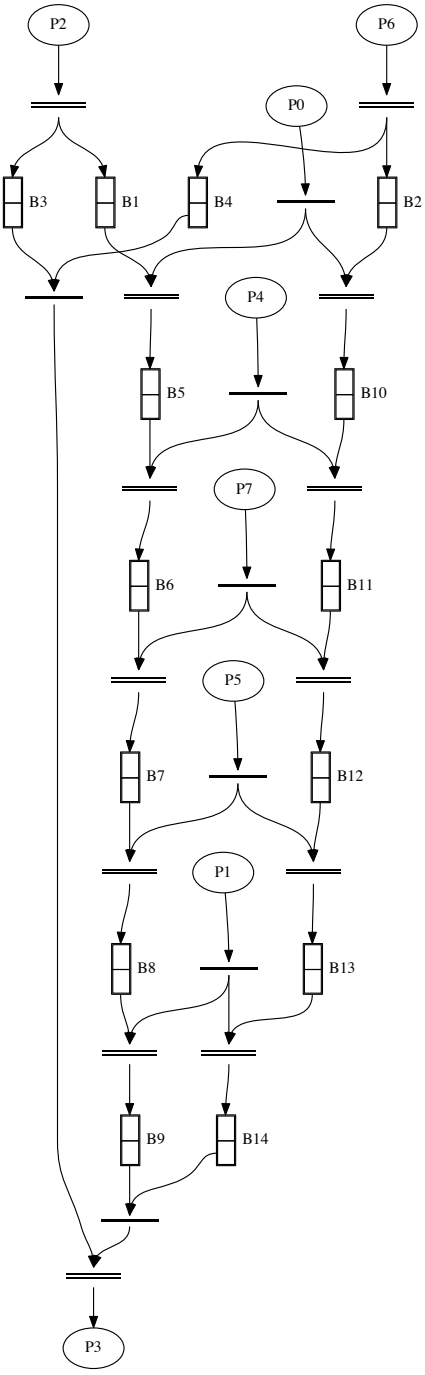



**Figure 7.7:** *Two entry scoreboard extraction result*

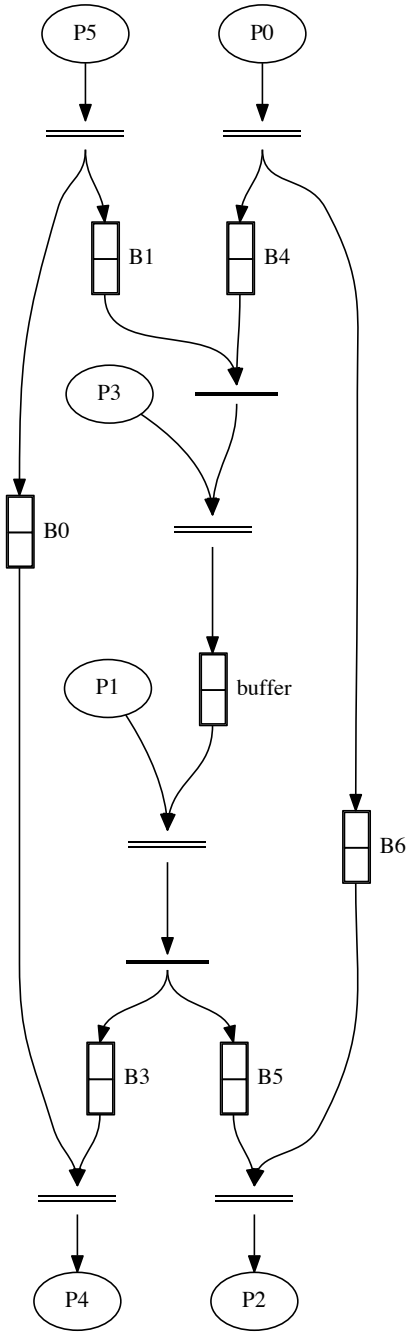



**Figure 7.8:** *Buffered virtual channel extraction result*

size of the SAT problem grows with the size of the network, but the difficulty does not. Estimating a linear growth in SAT solver runtime, we expect to see a $O(n^3)$ complexity, where $n$ is the number of repetitions of the buffered virtual channels. Measuring the runtime confirms this. Figure 7.9 shows the results for 1 to 10 repetitions.

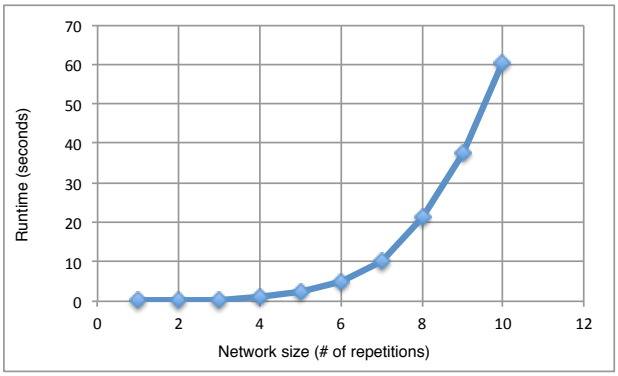



**Figure 7.9:** *Extracting xMAS from Verilog. Runtime for repetitions of Figure 7.1*

The runtime may be improved in two orthogonal ways. One way, is to build a cone of influence, or use a SAT solver that does so, such that as the network grows, the size of the SAT problem does not. This would give us an $O(n^2)$ runtime on average, for repetitions of the virtual channel. Another way to reduce the runtime, is to use the module structure in the Verilog code. This may not only reduce the runtime, but also opens up the possibility to create a similar module structure in the extracted network.

### 7.4.2 Validation of the resulting networks

Obtaining an xMAS model may provide some insight by itself, but the main reason to want an abstraction, is to be able to perform verification of the design using high level techniques. The method for finding inductive invariants (such as [Chatterjee and Kishinevsky, 2012]) is directly applicable to our generated graph. Such methods do not require specific xMAS properties like fairness of merges or the restriction that switching functions only depend on message data. Other techniques, like deadlock detection [Verbeek and Schmaltz, 2011b], do require such properties. In order to be able to use these techniques, we show that it is feasible to verify fairness properties using standard methods. The approach is as follows: we write a nuXmv [Cavada et al., 2014] description of the remaining flops. We hide the queue implementations (treat them as black boxes, leaving their outputs as undefined variables) in order to reduce the state space. We then formulate the property for a merge $A\langle x, y, t\rangle$ as an LTL sentence expressing that the merge is fair. That is: if $y$ gets turns (so not $\mathbf{G}\,\neg y_t$), and if $x$ requests turns ($x_r$), then $x$ should get turns as well ($x_t$). We want the same to hold when we interchange $x$

and $y$.

$$\mathbf{G}\,\mathbf{F}\,((x_r \to x_t) \vee \mathbf{G}\,\neg y_t) \wedge \mathbf{G}\,\mathbf{F}\,((y_r \to y_t) \vee \mathbf{G}\,\neg x_t)$$

The left side of that expression says that eventually $x$ gets a transfer if it is ready, given that $y$ is not blocking. The right side says the same for $y$. We check both sides of the expression separately using the model checker nuXmv. The runtime for nuXmv was well under a second. All merges provably satisfy the equation above, which does not come as a real surprise, as they also satisfy this property in the original design.

## 7.5 Conclusions

We have seen the algorithms of the previous chapters applied to several examples. The scalability of our algorithms show that generating invariants is relatively fast, deadlock verification has a larger and less predictable runtime, and extracting xMAS is the slowest of these methods. Arguably, the algorithms are ordered in how useful they are for verification. As such, the most useful method is the least scalable.

Ideas on how to improve the scalability are mentioned in the next chapter. This chapter also includes some small examples of where our methods fail.

# Chapter 8
# Discussion and conclusions

In the previous chapters, we have shown how circuits that describe communication fabrics can be analysed. Specifically, we have shown how to derive invariants that will aid in the analysis of networks. We have also shown that it is possible to prove deadlock freedom of networks directly, without suffering from the same type of state-space explosion problems to which model checkers are typically prone. Our approach, however, does have its limitations. These limitations fall into three categories:

1. Limitations concerning the performance or scalability of our approach. While we do not suffer from state-space explosion problems the way model checkers do, we do suffer from the fact that other parts of the algorithm have exponential growth with respect to the input.

2. Limitations concerning the analytical power of our approach. It is possible to prove or disprove a range properties. For a specific property, our approach is either able to prove it on some designs for which it holds (we refer to this as soundness), or to disprove it on some designs for which it does not (completeness).

3. Limitations concerning the feedback of our approach. It is nice to have something which can prove properties based on assumptions of certain networks. However, it is arguably even more important that whoever uses the approach, understands what these properties mean, and how to specify his assumptions.

## 8.1 Performance

Performance limitations are sometimes seen as limitations of least importance. One might say that if an algorithm is too slow for a certain purpose, we can either decide to parallelize it, use more or faster hardware, or precompute certain critical operations in memory. This section is not about the performance limitations to which such logic applies.

Certain designs suffer from performance bottlenecks that grow exponentially as certain features of that design grow linearly. These are bottlenecks that deserve the name 'limitation'. In such cases, algorithmic improvements will be necessary.

In verification of hardware, we always deal with code in languages that can express PSPACE-complete problems (and often even harder ones). Even after disregarding a lot of timing considerations, as we did in this work, we are left with NP-complete problems. This means that there will probably never be an algorithm with a polynomial runtime on a conventional computer to treat these problems. As a consequence, every available algorithm will at some point become prohibitively slow: as hardware grows in both speed and complexity, the increased complexity will always outweigh the increased speed in the long run. However, this does not mean that we cannot keep up with the growth of hardware. In fact, SAT and SMT solvers, integer programming solvers, and model checkers have shown exponential performance increases over the last decades as well. Their speedup has been significant.

It has in fact been significant enough to keep up with hardware: using todays algorithms on hardware from the 90's would be faster on most benchmarks than using algorithms from the 90's on todays hardware. But the speedup is more profound than that: in 1990 the number of transistors per microprocessor was about a million, it is roughly five thousand times higher these days (The Xbox One and the 61-core Xeon Phi, from 2013 and 2012 both report having 5 billion transistors).

We can be optimistic: algorithmic improvements for verification may be able to keep up with the growth of hardware. To achieve this, we can improve our algorithms using the same effort that improved SAT solvers over the last decades. Alternatively and more realistically, we can use SAT solvers for the bottlenecks and benefit from future improvements in their performance.

### 8.1.1 Limitations

The algorithm for expressing a Boolean value as an integer element – presented as a rewrite system in Section 4.1.4 – is an example of an exponential algorithm. As the logic depth is bounded in most hardware, this may not pose a problem. However, it is probably the most likely place to find performance benefits. Also, during the design phase, a design might be so different to the final hardware, that its logic depth is still too large, essentially growing with the size of the design.

To illustrate how the exponential growth may be a problem, we give an example with two switches. To determine whether the 'type' of a packet was a request or not, a switch would check if some field is equal to 0 (for request). Another switch would check if that field was equal to 1 (for response). These tests could of course be implemented by a single gate: both switches would have a small logic depth of just 1.

In a practical occurrence of this issue, however, the 'type'-field was modeled as an 8-bit integer. A bit-by-bit comparison between the value '0' and an 8-bit integer turned out to require a logic depth of 7. Combining the switch that checked for '1' with the switch that checked for '0' gave a logic depth of 14. While the final version of the two switches would probably have a low logic depth, it had a

much larger one during the design phase. This can make our analysis prohibitively memory consuming.

Another place where performance problems might arise, is when we propagate properties through queues. The integer representation of the output type is more or less 'copied' to the input of that queue. The complexity, expressed as the number of terms used to describe the type as an integer, is the product of the two.

We give another example for this. Suppose we have a large network, in which we consider two queues, $q_1$ and $q_2$. The output of $q_1$ is connected to a switch, which sends packets in which the first bit is high to $q_2$, and other packets to some other part of the network. The output of $q_2$ is connected to a switch as well, which looks at the last bit. Depending on the last bit, that switch sends packets to different parts of the network. For our analysis technique, we need to distinguish packets in $q_1$ based on the value of their first bit. Since some of these packets enter $q_2$, however, we also need to distinguish packets in $q_1$ based on their last bit. This means that for $q_1$, we will distinguish four types of packets. This shows that the number of packets that are to be distinguished may double every time we add a queue and a switch, another example of exponential growth in the memory requirements.

Such examples also arise in practice: in the case of a mesh, for instance, all of the destination address bits eventually end up being relevant in certain queues. This means that if there are 16 such bits, there will be $2^{16}$ packet types to consider. Of course, a mesh that uses 16 address bits might very well be a mesh with $2^{16}$ nodes, in which case this blowup is acceptable. Alternatively, a one-hot encoding could be used for the destination. This means that of the 16 bits, only one is high, which allows us to build faster switches. In such a case, a 16 bits address space would stand for only 16 nodes. Unfortunately, this would result in the same $2^{16}$ blowup, which in this case is not acceptable.

### 8.1.2 Possible improvements

There are several ways to improve performance. One way to improve the analysis, is to use more of the information from the underlying module structure. If a module is instantiated several times, the translation of one instantiation should still be very similar to the others. Currently, there hardly is any sharing between these translations. Some networks can have many identical parts. In a ring network, for instance, a packet could get a destination as the number of hops it needs to take: at each hop its 'destination' is decreased by one, and when it is zero, the packet leaves the network. Such networks can benefit directly from sharing translations. Our expectation is that this will speed up the analysis, and maybe even improve the quality of its results.

Other networks will have similar, but non-identical parts. For instance, a network in which the destination remains constant, and the packet leaves the network if its destination header matches the location of that specific node. In such cases, sharing translations would be more difficult.

Another way to improve performance, is by sharing more DAG nodes. Syntactically equal terms are identified in the underlying DAG. We can improve on this, by identifying all equivalent terms. This idea has already been applied to

AIGs in the model checker ABC [Mishchenko et al., 2005], where it has shown a great performance benefit. Being able to quickly test two nodes for equivalence can potentially improve our algorithms in several ways. The most obvious place where this could be used, is in the extraction of xMAS procedure. Other analysis procedures might also benefit, as a smaller DAG implies that fewer nodes need to be translated.

Similarly, we can improve the procedures in which invariants are required. In the translation from DAGs to a ring representation, every term is expanded until it is expressed in terms of free variables. As a consequence, two identical terms need to be expanded first, and then compared, even if they come from the same node in the original DAG. It should be possible to limit the number of terms that need expanding. This would overcome a very concrete bottleneck that arises in practice: To decide whether a certain packet header is equal to 0, all bits from the header need to be 0. If the expression to decide whether a packet is equal to 0 or not is not expanded, it can be represented as one term. The underlying logic depth for that comparison would then be irrelevant. This would decrease the memory usage, and boost performance.

Even better would be to reduce the need to expand terms. To decide whether or not a term needs to be expanded, we could look at its cone of influence. By using a SAT solver to decide whether two expressions can be true and false independently, we can even avoid having to expand terms altogether. This would give an algorithm that relies heavily on a SAT solver. We expect this to give a performance benefit. In general, an attempt to speed up our approaches by making better use of existing tools would be a new line of research in itself.

While building our implementation, we could often choose between implementing a new feature, or speeding up the existing ones. Unless the performance of the implementation was prohibitively slow, we decided on the former. Implementing features that increase the class of designs that can be analysed is often the most exciting, and the most rewarding choice. As a result, we believe that there is still a lot of low hanging fruit to improve the performance.

In contrast to improving the performance, we can also imagine many ideas that would add to the strength of our analysis. Adding these may degrade the performance, but adds to the analytical power of our methods. This will be discussed in the next section.

## 8.2 Analytical power

We can prove deadlock freedom of several designs. For certain designs for which our implementation fails to prove deadlock freedom, we are able to indicate how to change the implementation. Unfortunately, none of these techniques will be able to prove deadlock freedom for all designs. The fundamental reason is that deciding whether arbitrary RTL is deadlock-free, is a PSPACE-hard problem. We have tried to abstract away from issues that make it PSPACE-hard, and obtained performance benefits from it.

Note that we do not claim that verification of hardware is PSPACE-complete. This depends on how the hardware is described exactly, and the property being checked. Most, if not all, hardware descriptions that translate into an arbitrary

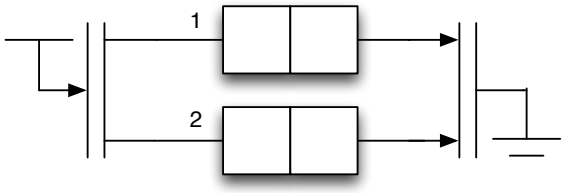



**Figure 8.1:** *Two parallel queues.*

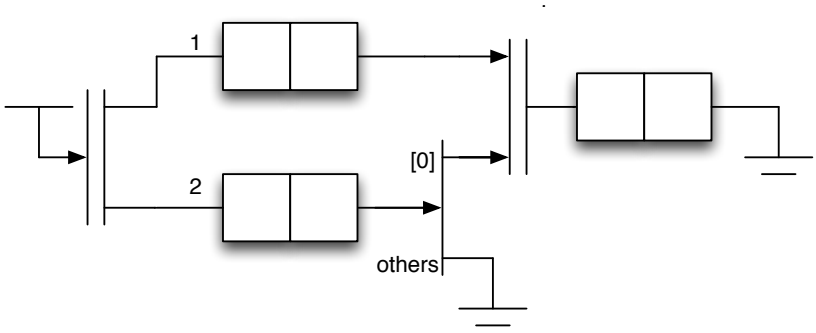



**Figure 8.2:** *Two parallel queues and one switch.*

number of gates, make the problem harder. For example, the reachability problem on hardware described with word-level constructions is shown to be EXPSPACE-complete [Kovásznai et al., 2014].

There is a trivial way to make all our analysis techniques sound and complete for a certain problem. We can simply make a call to a model checker in those cases where our techniques would normally be unsure. In such a case, we do not expect the model checker to be very efficient.

This would mean that we use the efficient algorithms for the cases we considered 'solvable' so far, but resort to a prohibitively slow analysis for the cases we believe to be corner cases. This would only give a false representation of our analysis: experimental results would give good results as the slowest parts of the analysis are never run.

So far, we did not connect any model checkers to our analysis: a corner case network will not result in a timeout, but simply in an answer which is probably unsatisfactory to the user, such as a candidate counterexample to liveness in a live network.

This section will look at examples for which our analysis techniques do not give the desired result, for which we could not find a fundamental reason to 'miss' these cases. Future techniques may be able to efficiently solve these cases.

### 8.2.1 Limitations

We proved that any linear invariant that holds inductively, will be found by our analysis technique of Chapter 4. This assumes the availability of variables over

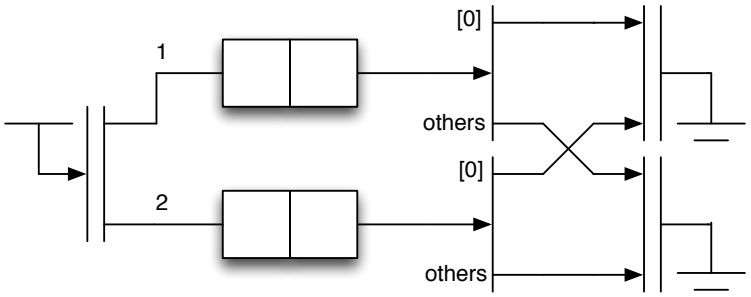



**Figure 8.3:** *Two parallel queues and two switches.*

which the invariants are calculated, and full independence of these variables. In some examples, neither is the case. As a result, too many false deadlocks may be found in the analysis of Chapter 5.

As an example of a network in which not all variables are available, see Figure 8.1. The obvious invariant – which is found – is that the number of packets in the first queue, say $q_1$, is equal to $q_2$, the number of packets in the second. Additionally, the packets entering and leaving the first queue are the same packets as those entering and leaving the second one. As a result, the number of packets of type 0 in the first queue, say $q_1[0]$, is equal to those in the second, say $q_2[0]$. This second invariant is not found for two reasons. The first reason is that the variables $q_1[0]$ and $q_2[0]$ are never created, because nothing in this circuit depends on the type of either of the packets. This is what we mean with the availability of variables: they are never a candidate variable, so they do not appear in any invariants.

We can slightly change the circuit such that the variables $q_1[0]$ and $q_2[0]$ do become available, see Figure 8.2. This figure shows two parallel queues between a fork and a join, with a switch after the join. The join just concatenates the data of both packets, and the switch will route packets upwards if both of the concatenated packets are of type 0. An extra queue after the last join makes sure that the type of packets in $q_1$ and $q_2$ becomes relevant. Unfortunately, the invariant $q_1[0] = q_2[0]$ is still not found. This happens because events in which packets of type 0 leave the two queues cannot happen independently, but our analysis assumes this can. The invariant $q_1[0] = q_2[0]$ relies on this. If we could somehow get the packets from these queues out-of-order, the invariant $q_1[0] = q_2[0]$ would not hold.

Even if the right invariants are found, it may still be impossible to prove liveness of the circuit. Figure 8.3 is an example of a circuit which, given that all sinks are fair, has no local deadlocks. It has two queues in parallel, each followed by a switch that routes packets of type 0 upwards, and others down. Packets are then synchronised, because of this join it becomes obvious that in the event in which a packets of type 0 leaves one queue, a packet will always leave the other queue simultaneously. As a result, $q_1[0]$ and $q_2[0]$ are created, and the required invariant $q_1[0] = q_2[0]$ will be found. Unfortunately, it is not immediate that the packet at the head of the first queue will always have the same type as the packet at the head of the second queue. As a result, we find a deadlock configuration which

would never occur, in which each queue holds two packets of a different type, and in a different order.

A final limitation to our method lies in that we assume a single clock for all components. In practice, networks may have multiple clocks. In such cases, we would not be able to describe queues in terms of their in- and output transfers. This means that we cannot use our techniques to perform any analysis on such circuits.

### 8.2.2 Possible improvements

Accounting for multiple clock domains is most likely the easiest limitation to lift. Clock domains are usually sufficiently isolated from one another: input and output events, or the averages as in Chapter 5, from different clock domains can be assumed to be incomparable.

Accounting for the order between packets may be very challenging. However, the relevant information may be a high level property that is expressible in terms of the wires in the RTL. In the example of Figure 8.3, we could add the information that the packet at the output of the queues is always the same. This extra piece of information would even suffice to get the additional invariant that is relevant in Figure 8.2. Finding a way to express such a property, or perhaps even derive it, would cause our methods to be applicable to a greater class of circuits.

One way to arrive at the relevant high level properties, would be to develop a tight integration with model checkers that use property directed reachability or IC3 [Bradley, 2011]. The idea of this model checking technique is that it guesses invariants based on whether certain states can be reached or not. These invariants are typically of a very different nature than the invariants we find: they are typically Boolean formulas which are one-step inductive. The invariants we find are linear formulas. It is already known that adding the linear invariants which we can derive, will help with the efficiency in model checkers. We believe that using the Boolean invariants found by the model checkers, will help find more linear invariants. Seeking improvement in this direction would be an interesting line of future research.

Finally, we could search for polynomial invariants, instead of linear invariants. The procedure we use to find linear invariants uses a certain normal form to find them: the Jordan normal form. For polynomials, such a normal form exists as well, called the Gröbner basis. We do not know whether this would yield stronger invariants, or more efficient techniques to find them. These techniques are scalable enough to verify large arithmetic circuits, as shown by Farahmandi and Alizadeh [Farahmandi and Alizadeh, 2015]. So far, we have not found non-linear polynomial invariants, so we do not expect direct improvements in the analysis of NoCs. Using these techniques could, however, be a first step to make our techniques applicable in other domains.

## 8.3 Feedback limitations

We can say that a method is ‘practical’ as soon as someone can apply a method to his or her own real world examples. So far, we have been the only people using

our tool, Voi. Consequently, we assumed that those using it, would know what they were doing. Even though we kept the required interaction with our tool to a bare minimum, mistakes can be made along the way. This section can be read as a warning and disclaimer section, but ideally any tool would alert users on all errors they might make. As such, any error that can be made accidentally, should be regarded as a limitation to our tool.

Some of these issues can be resolved through better feedback to the user. Other issues require extra checks, which would be more involved but also more interesting.

### 8.3.1 Limitations

In terms of user feedback, the biggest issue is with our approach in Chapter 5. Here, an SMT solver will say SAT or UNSAT, from which we know that a NoC potentially has a deadlock, or that it is deadlock-free, respectively. Both cases turn out to be unsatisfactory: UNSAT can be an unsatisfactory answer, since it provides no additional information. Although this would prove absence of deadlocks, it does not explain why there is no deadlock. Some of these reasons would indicate that errors may still occur in the network: the specification of queues might be wrong. Packets may be misrouted. Packets may be dropped before they reached their destination. In other words: deadlock-free designs may still contain errors. In these cases, receiving 'UNSAT' is unsatisfactory.

If the NoC potentially has a deadlock, our solver will return SAT, and we would usually like to know more about it. Questions would be: how could this deadlock arise, and what does it look like? Given that all queues in the NoC are identified, a deadlock could be visualised by showing the packets inside queues. Similarly, we can imagine that a trace could show where the packets are over time. Unfortunately, reconstructing such a trace from the model is not trivial.

Another problem in the 'SAT' case, is that a potential deadlock does not guarantee that there actually is a deadlock. To investigate the root of the problem, assumptions about the design may need to be made in order to rule out any deadlocks. Such assumptions may or may not hold, and it could be helpful to have an overview of these assumptions, such that they can be checked later. One of the assumptions which is made in any case, is that all black box modules match their pre-conditions. In other words: the specification of queues might be wrong. However, in investigating a design, whether it is faulty or not, it should be possible to make additional assumptions, and check them later. Currently, it is not possible to add assumptions, nor is it possible to check any of the assumptions made by default.

Finally, other tools and verification procedures might require the result of our analysis. If a design is proved correct in, say, a theorem prover, we could add the deadlock freedom (under the relevant assumptions), to it as an axiom. This would allow for errors to arise in many places: the theorem stated in the theorem prover might differ from the one we actually proved, the assumptions we made while implementing our algorithms may not hold, additional assumptions may be forgotten when adding the theorem to the prover, we might have made errors in the implementation of our analysis, and the design under investigation in our method

may be different from the design under investigation in the theorem prover. If the design changes during its verification – which is likely – then there is a big risk of accidentally using a deadlock freedom proof of one design, while assuming that the proof concerns another.

### 8.3.2 Possible improvements

Ideally, our tools would be a part of larger frameworks. Depending on the limitation mentioned, a different framework would apply. If we want to visualise a deadlock, a design automation tool in which the user can jump between visualisations of the network, wire values and Verilog code, would be ideal. If we want to investigate how a deadlock arose, integration with model checking techniques might be helpful. Finally, if we want to be able to depend on the results of our tool, while combining them with other results, integration into a theorem prover would be ideal. These three frameworks: design automation tools, model checkers, and theorem provers, all exist presently. We have not integrated our tools with either of these.

To integrate with a design automation tool, we would need to build the visualisation of the deadlocks. There are several design automation tools, and each requires a different means of integration. We do not have a clear understanding of the work required to carry out such integration.

For a model checker, there are a few open source candidates, the most important one being ABC [Synthesis and Group, 2015], as it has a strong focus on hardware. Another choice would be nuXmv [Cavada et al., 2014], which we used in Chapter 6 to verify properties about the extracted network. The use of these would require us to export and import properties through an interface for model checkers. This would require an acceptable amount of work. To help the model checker find a trace in an acceptable amount of time, would require more research. The information from the SAT instance should be able to help direct its search.

Integrating our method with a theorem prover could be done in several ways. First, we could define our tool Voi, or a version thereof, in a theorem prover, and prove their correctness. This would allow us to run the algorithms in that theorem prover. To do so would require a very large amount of work, and degrade the performance and flexibility of Voi. Another way would be to let Voi emit certificates, which should be checkable by the theorem prover in a reasonable amount of time. Checking such a certificate should cause the theorem prover to accept the correctness of a network as a theorem. This would also require some amount of future research, but largely preserve the performance and flexibility of Voi. Either way, all assumptions will have to be made explicit in order for the theorem prover to accept the theorem. Another way to integrate our method with a theorem prover, would be to allow our correctness proofs to be exported as an axiom. This would be the easiest to achieve, but also the most dangerous: any assumption we forget to export would strengthen the axiom, in which case it no longer holds.

The feedback limitations illustrate that Voi has been developed as a proof of concept.

## 8.4 Conclusions

We verify properties about interconnects from their low level implementation. Instead of requiring high level knowledge about the design, we only use a specification of the interface of queues in the design.

The gate level implementation of a NoC design can be translated to combinational logic. We can automatically derive linear inductive invariants from the implementation and a specification of the interfaces of queues and registers. Similarly, we can use the same information as input to an algorithm that formulates a Satisfiability Modulo Theory (SMT) problem that has a solution if the NoC has a reachable local deadlock. For many NoCs, the SMT problem turns out not to have an answer, which proves that these NoCs are live. To reproduce an abstract-level description from the implementation, we give a procedure that constructs a tree of synchronising and arbitrating elements from the interface descriptions. After orienting the elements in the tree, we obtain an xMAS-like network that is transfer equivalent to the implementation. Most of the proposed algorithms are implemented as part of a tool called Voi.

# Appendix A
# Verification of Interconnects: an Implementation in Haskell

This chapter explains how reasoning about circuits is implemented in Haskell. It is intended for who is interested in some technical details.

The Haskell implementation provides a tool that allows for different verification techniques. We have called this tool 'Voi'. The tool is illustrated in Figure A.1. The dotted rectangles indicate in which file a certain step can be found primarily. Arrows indicate the flow of information and rounded rectangles are data-structures. Note that the boxes 'Liveness verification' and 'BooleanXZ to SMT' are only drawn to indicate where these parts would fit in the tool. As indicated in Chapter 4, they are not a part of Voi yet.

After processing a Verilog file, a directed acyclic graph, or DAG, is constructed to describe each module. The main code for this can be found in 'Processor.hs'. These modules are simplified and combined; the resulting module is described as a DAG again, as indicated in the figure by the rounded rectangle called 'Top level module'. All verification techniques presented in this thesis can be integrated into our tool by creating an appropriate instance of a `BooleanXZ` class. This class provides a uniform way to handle four-valued logic. There is a function to translate the DAG structure into an arbitrary instance of the `BooleanXZ` class.

A verification technique can be seen as something which performs these steps:

1. Parse a Verilog module into a DAG.
2. For all the instances in that module: substitute their simplified DAGs in a manner similar to the simplification step.
3. Simplify the module such that every wire has its symbolic fixed-point value.
4. Translate the symbolic representation into another one, through the interface of the `BooleanXZ` class.
5. Perform the analysis on the representation obtained in this way.

Most of the experimentation happens in the last two steps, which are extremely flexible.

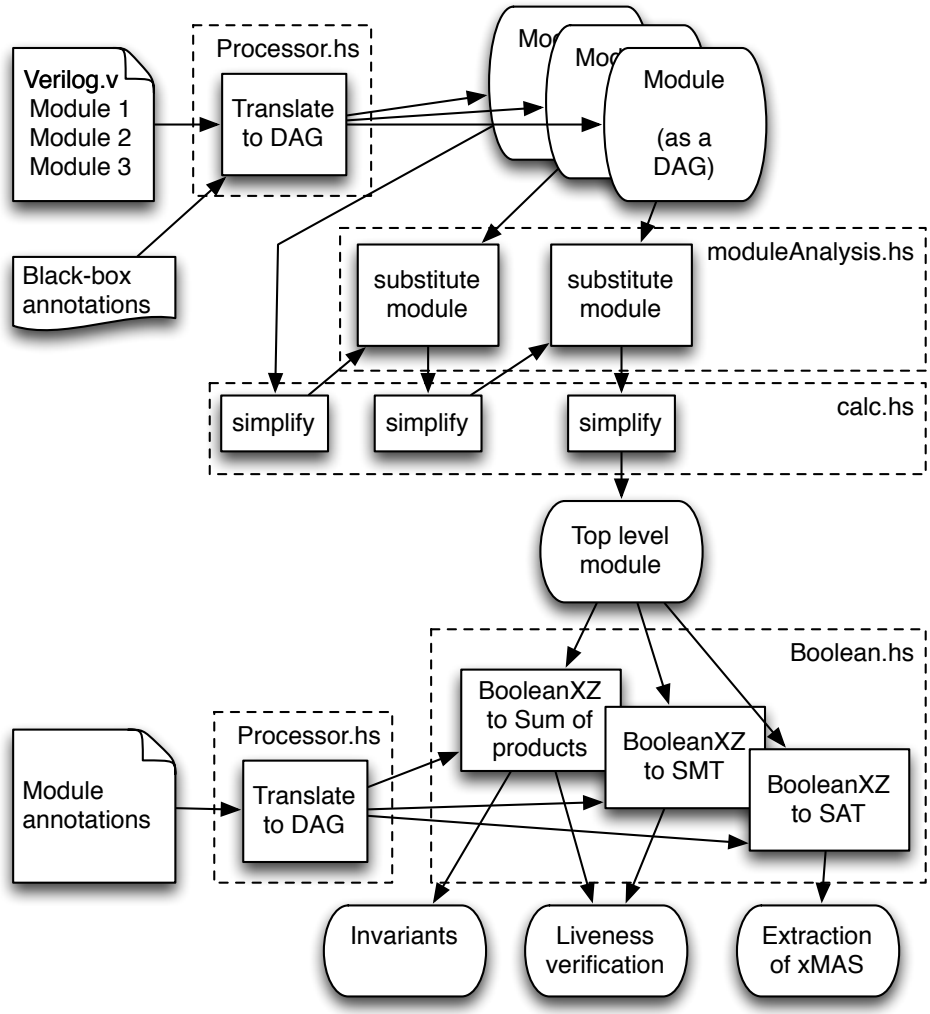



**Figure A.1:** *Voi tool as implemented in Haskell*

We will give a general introduction about DAGs in Section A.2. We then turn to the specifics of our `BooleanXZ` class, and show that we can make a symbolic instance for it using DAGs in Section A.3.1. This symbolic representation is used for instantiating and simplifying modules. The way modules are simplified is illustrated in Section A.3.2. We end this chapter with an abstract representation that can be used for Booleans, namely that of a ring, in Section A.4.2. This will illustrate how one of the analysis techniques can be used in our tool.

## A.1 Example input: a design in Verilog

There are clear benefits of using four-valued logic. Circuits that make effective use of un-driven wires may require less on-chip area and use less power, when compared to designs that do not use such wires. We believe that the main reason that four-valued logic is scarce in practice, is that formal verification of circuits using it is not properly supported. Nevertheless, such circuits do occur, especially at the boundary of chips, where io-ports are common.

We constructed a small design, to serve as an example about what details need to be supported for the analysis of circuits. It is inspired by a bus structure, but it adds a delay, repeating data in a certain direction. Such a repeater can be useful for decreasing the logic depth of a circuit, thereby increasing clock speed. It could also help to break long wires, to get a better signal to noise ratio. To repeat data, registers are required to store it. Such registers can be expensive, so we will use only one register per data bit. For simplicity, we only use a one bit wide data size in our design, and indicate which part has to be repeated per bit.

For this section, we use an example that may be useful in communication fabrics. We call our module `TwoWayRepeater`, and show it in Figure A.2. The gate-level verilog code for this module is:

```
module TwoWayRepeater (clock, reset, fromLeft, fromRight,
    enabled, dataLeft, dataRight);
 input clock,reset; // for the register
 input fromLeft; // direction of data flow
 input fromRight; // don't write data when False !
 input enabled,dataLeft,dataRight;
 wire writeData,readData; // data from/to register
 wire outData; // output of register
 wire wasEnabled,wasntEnabled,leftToRt,rightToLt;

 nor(leftToRt,rightToLt,fromLeft);
 nor(rightToLt,leftToRt,fromRight);
 reg_module_no_reset data_register (.drive(writeData), .clk
    (clock), .q(outData));
 // for each bit of data, the following has to be repeated
 reg_module was_enabled_reg (.drive(enabled), .clk(clock),
    .set(1'b0), .reset(reset), .q(wasEnabled), .notQ(
    wasntEnabled));
 // using both a pmos and an nmos increases reliability
```

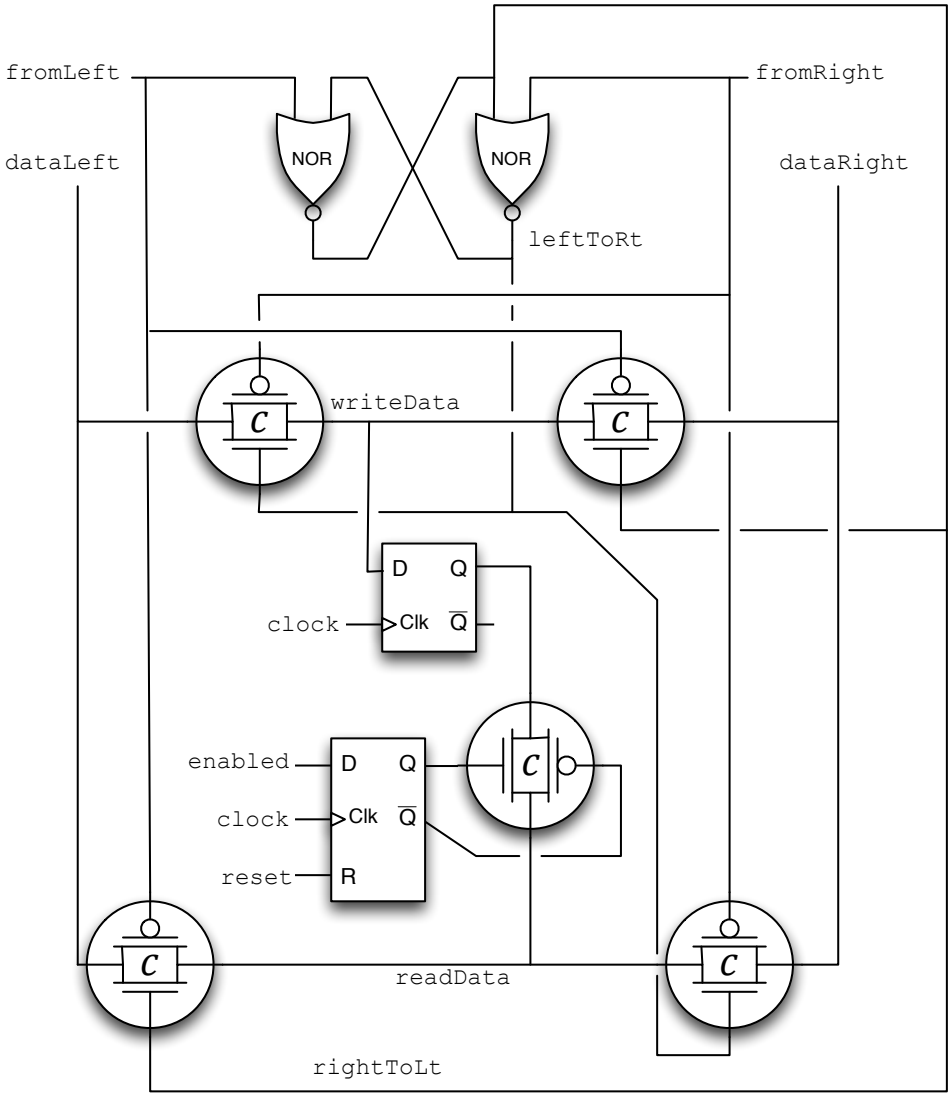



**Figure A.2:** *A two way repeater*

```
 // cmos is the gate-name of these two transistors:
 cmos(readData , outData , wasEnabled, wasntEnabled);
 cmos(writeData, dataRight, rightToLt, fromLeft);
 cmos(writeData, dataLeft , leftToRt , fromRight);
 cmos(dataRight, readData , leftToRt , fromRight);
 cmos(dataLeft , readData , rightToLt, fromLeft);
endmodule
```

The design has two task: The first task is to take data from 'left' to 'right'. The wire `dataLeft` is driven (with data), and the wire `fromLeft` is set to high (otherwise, it is kept low). In this case, `leftToRt` is high, the data is put at `writeData`, and written to the register called `data_register`. Note that we use a `cmos` gate to set the value of `writeData`. In hardware, this gate connects `dataLeft` to `writeData` bi-directionally if `fromRight` is low, and if `leftToRt` is high. This uses two transistors, one of them is a `pmos`, which connects `dataLeft` to `writeData` if `fromRight` is low. If the data-signal `dataLeft` is low, however, the `pmos` gate becomes resistive. This is undesired analog behavior, that can cause the (analog) value of `writeData` to slightly differ from the zero voltage supplied at `dataLeft`. The `nmos` transistor suffers from a similar problem, but only when `dataLeft` is high. To solve this problem, we use both transistors.

If `enabled` is high, the next clock tick will put that same data at `readData`, which is connected to `dataRight`. The second task is to take data from 'right' to 'left', and it is similar to the former. Before changing directions, `enabled` has to be low for one cycle. This allows the last data to be read.

In the case that the repeater is not used, `fromLeft` and `fromRight` are both low, and `dataLeft` and `dataRight` are connected via the `pmos` transistors and via both the `writeData` and `readData` wires. It is therefore important that no data is written from the left while `fromLeft` is low, and no data is written from the right while `fromRight` is low. Our design contains a cycle between two wires (`leftToRt` and `rightToLt`). By adding that cycle, the design is no longer combinational. The distinction between `leftToRt` and `fromLeft` has the benefit of closing all `cmos` gates when data is written from left and right simultaneously. Using one-input `not` gates instead of the `nor` gates would achieve the same effect, use fewer transistors, and avoid creating cycles. The only reason we add the cycle, is to show how we obtain an acyclic network for analysis. If the inputs `fromLeft` and `fromRight` both are low, the wires `leftToRt` and `rightToLt` retain their old values. For performing sound analysis, the value of `leftToRt` and `rightToLt` can be over-approximated with the unknown value `X`. We use this example because it illustrates that the circuit can be in a valid state and still have an `X` value occur due to the sequential nature of the design.

The design uses ten transistors per data-bit, one (non-resettable) register per data-bit, plus a fixed cost of twelve transistors and one additional register. Since our parser does not handle Verilog parameters, we just give the implementation for one bit of data. Our implementation parses most gates in the Verilog standard, but we will just use transistor-level Verilog in our example. Even this simple transistor-level Verilog is not supported in all tools. Specifically, some FPGA architectures may not have support for them. The effect of internal tri-state buffers are said to

sometimes cause problems during simulation [Coffman, 1999].

We mentioned in the previous chapter that the Verilog standard assumes `pmos` and `nmos` to be uni-directional (the output being the first argument). We make the same assumption in our tool. There is quite a large collection of switch primitives doing the same thing in Verilog: `pmos`, `tranif0` and `bufif0` (or `nmos`, `tranif1` and `bufif1` respectively) are described as doing the same thing. We propose the following distinction:

`tranif0`. The designer intends bi-directionality: this should be synthesized as a symmetric transistor. The code described in this work assumes bi-directionality.

`pmos`. The designer intends uni-directionality, and guarantees this gate will be used that way, even when synthesized as a symmetric transistor. The code described in this work assumes uni-directionality, as if it were a `bufif0` gate, but future versions may give warnings or even fail with an error message when this condition is not satisfied.

`bufif0`. The designer intends uni-directionality, but does not guarantee that a symmetric transistor will behave that way. A synthesis tool should add an extra `buf` gate if needed (or rather: assume it is needed, and optimize it away if possible). In this view `bufif0` does not model a single transistor. Our tool therefore must assume uni-directionality: bi-directionality would be wrong.

As such, we chose the `pmos` gates for our design (combined with their `nmos` counterparts). We know the direction of information flow, but want full control over the number of transistors used.

The example contains a lot of the Verilog that is considered 'tricky' to deal with. Delays and wire-strengths, however, are not present in our example, since our tools do not support these features.

## A.2 On using DAGs in Haskell to represent formulas

Before diving into the details of our Haskell implementation, it helps to get some background into how a DAG can be implemented. This section will be just about that, in the hope that it aids in understanding the implementation. If you feel confident about DAGs and Haskell, this section can be skipped without affecting the flow.

The code in this section forms a module called "MyDAGS". Code meant as an illustrative example not part of this module is shown without syntax highlighting. Code that is part of that module is highlighted as follows:

```
{-# OPTIONS_GHC -Wall #-} -- show almost all warnings
{-# LANGUAGE RankNTypes, GeneralizedNewtypeDeriving
           , RoleAnnotations #-}
module MyDAGS (main, WithDAGT, runDAG, freezeRefN, addNode,
```

```
        showDAG, RefN, Node, DAG) where
import Control.Monad.State
```

This code begins our Haskell file with the necessary switches, imports and exports. If the reader wishes to follow this section using Haskell, highlighted portions can be copied into an editor, while the non-highlighted ones can be tried from `ghci`.

A DAG is a set of nodes together with a function from each node to a list (or set, depending on who defines "DAG") of nodes, in such a way that the function forms no cycles.

The "no cycle" requirement means that we will get an invariant which we need to keep consistent. This involves keeping track of which DAG each node belongs to. To do this, we add a type variable to each DAG, node and reference.

```
newtype DAG name = DAG [Node name] deriving Show
data Node name = Node String [RefN name] deriving Show
newtype RefN name = RefN Int deriving Show
-- add a node to the DAG, return reference to the new node
hiddenAddNode :: String -> [RefN n] -> DAG n
              -> (RefN n, DAG n)
hiddenAddNode label refs (DAG nodes)
 = ( RefN (length nodes)
   , DAG (nodes ++ [Node label refs])
   )
```

The `newtype` statement is nearly synonymous[1] with the `data` statement. In the first statement, the first occurence of `DAG` is a type, and the second is a constructor. The `deriving Show` is not necessary, but convenient for debugging, and presenting the results below.

Note that the `name` part is not used anywhere, this is called a Phantom Type, and it will help us keep track of nodes and which references belong to which DAGs. If we refrain from using cyclic definitions, then "new" nodes can just be built from "old" ones. Since the old ones point to existing nodes, the new nodes will point backwards in the latest DAG:

```
(dagnode1,dag1) = hiddenAddNode "node1" []          (DAG [])
(dagnode2,dag2) = hiddenAddNode "node2" [dagnode1] dag1
-- dag2 == DAG [Node "node1" [],Node "node2" [RefN 0]]
```

Unfortunately, Haskell will just match "name" to any type we throw at it, so we could also write:

```
(dagnode1,dag1) = hiddenAddNode "node1" []          (DAG [])
(dagnode2,dag2) = hiddenAddNode "node2" [dagnode1] (DAG [])
-- dag2 == DAG [Node "node2" [RefN 0]]
-- ouch! Node 0 refers to Node 0, a cycle!
```

[1] Their difference is in a property called strictness.

### A.2.1 Use of RankNTypes

So here is the problem we wish to solve: `"node2"` points to itself, and we only used `hiddenAddNode` and `DAG []`. The problem is that the second `(DAG [])` refers to an old version of the DAG, while `dagnode1` is a reference to a newer version. As a solution, we pass the `DAG` implicitly, such that we are forced to use the newest version all the time. Monads are great for this; we should be able to write:

```
someDAG :: String
someDAG
  = showDAG (do dagnode1  <- addNode "node1" []
                _dagnode2 <- addNode "node2" [dagnode1]
                return ())
```

The type of `DAG n -> (DAG n, RefN n)` is precisely the right type for the state monad. The state monad also allows us to get to the internal state via `get` and `put`, which is what we want to prevent. Therefore we create our own monad instead, and use it for addNode:

```
newtype WithDAGT name a = WithDAG (State (DAG name) a)
   deriving (Monad,Applicative,Functor)
addNode:: String->[RefN name]->WithDAGT name (RefN name)
addNode label refs
  = WithDAG (state (hiddenAddNode label refs))
```

The `deriving` statement gives us the most reasonable instance for the `Monad` class, which is by definition also an instance of `Applicative` and `Functor` classes.

We have to supply an initial empty DAG somewhere, somehow. This is where rankNTypes come in. We use them with exactly the purpose they were introduced for in 1994 [Launchbury and Peyton Jones, 1994].

```
runDAG :: (forall name. WithDAGT name a) -> (a, DAG ())
runDAG (WithDAG st) = runState st (DAG [])
```

Function `runDAG` allows us to get all the values. The 'name' of the corresponding DAG, however, is marked as `()`. This prevents us from ever using that DAG inside `runDAG` ever again. This is because `name` does not match with `()`, which is exactly what that `forall` was for. In fact, it even prevents us from getting references out of the `WithDAG`:

```
someNode = fst (runDAG (addNode "node1" []))
```

gives the following type error:

```
Couldn't match type 'a' with 'RefN name'
  because type variable 'name' would escape its scope
```

There is an easy fix:

```
freezeRefN :: RefN name -> WithDAGT name (RefN ())
freezeRefN (RefN i) = return (RefN i)
_someNode :: RefN () -- returns "RefN 0"
_someNode = fst (runDAG (do node<- addNode "node1" []
                            freezeRefN node))
```

Once again, we get a reference which we will never be able to use inside `runDAG`. This means that `runDAG`, `freezeRefN`, and `addNode` are perfectly safe to export. We are able to write `someDAG` with them, since we can implement `showDAG` with just `runDAG`:

```
showDAG :: (forall name. WithDAGT name a) -> String
showDAG wd = show (snd (runDAG wd))

main :: IO ()
main = putStrLn someDAG
-- output: DAG [Node "node1" [],Node "node2" [RefN 0]]
```

When using this as a library, we would export `RefN`, `Node` and `DAG`. These are just type names, which is fine, as long as we do not export their constructors: we want all construction inside `WithDAGT`. If we would export the constructors, it would then be trivial to cast a `RefN name` to a `RefN name2`.

### A.2.2 Does type safety protect against all cycles?

We showed that certain programs that would introduce cycles in a DAG, are no longer allowed by Haskell's type system. We can ask whether this is condition is sufficient, that is: whether we can be certain that our DAGs are acyclic, independent of the code. The answer is no: there are at least two ways to introduce cycles. The "van-Laarhoven DAGs" provide a data-structure that does a better job at preventing these cycles [Van Laarhoven, 2012], but their types make them much harder to use in practice. We will show how to introduce cycles in DAGs, so we can avoid using the tricks that introduce them.

The first way to introduce a cycle, is by using `coerce` from the module `Data.Coerce`. This is a recent feature in Haskell, and we can prevent its use on our DAGs. However, there is another version, called `unsafeCoerce`, which has been available for longer and cannot be disabled that would yield the exact same results. Unsafe coercions will always allow us to circumvent the type system, even when we would use van-Laarhoven DAGs. This example shows how we can do the same with `coerce`, and justifies that we disable this. With `coerce`, we can change the data type of a node:

```
someNode :: forall name name2. WithDAGT name2 (RefN name)
someNode = do dagnode1 <- addNode "nodeX" []
              return (coerce dagnode1)
freeNode :: forall name. (RefN name)
freeNode = fst (runDAG someNode)
```

We were only supposed to be able to get nodes of the type `RefN ()`, such that we could not re-use them, but our use of coerce changed this. We can use this to create a 'wrong' DAG:

```
someDAG2 :: forall name. WithDAGT name ()
someDAG2 = do _dagnode  <- addNode "node1" [freeNode]
              return ()
```

Another way to introduce a cycle, is to use Haskell's lazy evaluation strategy. A program which 'clearly loops' in some languages, might terminate in Haskell. The textbook-example being `take 5 [0..]`, which produces a list with numbers from 0 to 4 in finite time, even though the argument `[0..]` cannot be fully computed. Here is how we can use that to introduce a cycle:

```
someDAG3 :: forall name. WithDAGT name ()
someDAG3 = do build
              return ()
 where
     build = do builtNode <- build
                dagnode <- addNode "node1" [builtNode]
                return dagnode
```

In this case, the program loops on `builtNode <- build`. It does not ever produce the erroneous node, which – considering our objective to prevent it – is fortunate. We would have to change the implementation of `WithDAGT` and its Monad instance, to make `someDAG3` productive. The point of this example, however, is to show that the type system does not prevent us from writing such programs, even if the runtime system does.

Nevertheless, we found the type restrictions sufficiently strong. The counterexamples we attempted to produce here are not programs we expect to accidentally write in practice: we can disable 'coerce' and avoid 'unsafeCoerce'. Additionally, it is desirable and natural to write terminating programs, so our second example can be avoided as well.

## A.3 Representations of four-valued Booleans

This section will present portions of Haskell code that are excerpts from our Voi tool. For brevity and readability, some parts of the code are not documented in this thesis. The parts that are shown were selected to illustrate how different circuit analysis techniques can be combined.

We separate the use and the implementation of Boolean formulas through a class. In the previous section, we saw how `addNode` should be used within a Monad. For that reason, our class `Boolean` has functions of that form:

```
class (Monad m, Applicative m) => Boolean m boolean where
 fromBool :: Bool -> m boolean
 andC,orC,eqC,impC,notImpC,xorC
  :: boolean -> boolean -> m boolean
```

```
 iteC :: boolean -> boolean -> boolean -> m boolean
 notC :: boolean -> m boolean
```

This class basically contains all the two-valued logic gates we encountered, with some extra functions which were useful in some simplification procedures. Note that the class could be much smaller: just `iteC` and `fromBool` are sufficient to define all of the other functions. Although this is not shown here, we provide default function implementations (which can be overwritten when desired) to alleviate the burden of defining these. Note that this class can be used for Boolean functions, but also for the four-valued logic gates used in circuits. To handle circuits, some of the four-valued gates are missing. They are provided through separate class, called `BooleanXZ`:

```
class (Boolean m boolean) => BooleanXZ m boolean where
 gbufC, combineC :: boolean -> boolean -> m boolean
 bufC :: boolean -> m boolean
 combineListC :: [boolean] -> m boolean
 combineListC [] = zC
 combineListC [a] = return a
 combineListC (h:lst) = foldrM combineC h lst
 xC, zC :: m boolean
```

This class provides the gate 'buf' (a gate to direct the flow of information), its gated version 'gbuf' (when its first argument is true, it acts as 'buf'), and the operation required to combine wire values: `combineC`.

### A.3.1 Symbolic instance

There is a literal translation from the `Boolean` and `BooleanXZ` classes, to a data structure, but we do some grouping according to how the operators can be treated. Instead of creating a separate constructor for each Monoid operation (`AND`, `OR`, etcetera), we create one constructor `MSymbolic`, which gets the relevant operator as its argument of type `MOperator`. Similarly, none-monoid operations are grouped under the constructor `Symbolic`, and unary operations are grouped as `Unary`. The if-then-else operation is the only ternary operation, so it is treated separately. While `X` has its own constructor, the monoid operator to connect wires, called $\cdot$ in Chapter 3, can be used to represent `Z`, as `MSymbolic CONNECT []`.

```
data Symbolic a
 = Symbol ScopedWireName
 | Symbolic Operator a a
 | MSymbolic MOperator [a]
 | Unary UnaryOperator a
 | ITE a a a
 | X
 deriving (Foldable, Functor, Traversable, Show, Eq, Ord)
```

The nodes in our DAG will be of type `Symbolic Int`. The `Int` is kept parametric here, which can help prevent us from mixing up different references

in certain functions. Since `Symbolic` is not exported, we do not need to add a phantom type for it. References will be called `WireReference`, and they do have a phantom type. The `ScopedWireName` consists of the name of a wire, a list of instances to indicate its scope, and an integer as its hash for faster sorting.

```
data ScopedWireName = ScopedWN Int [ByteString] ByteString
                      deriving (Show, Ord, Eq)
newtype WireReference a = WireRef{runWireRef::Int}
type DAG = Sequence.Seq (Symbolic Int)
newtype WithDAG btype a
 = WD (State DAG a) deriving (Monad, Applicative, Functor)
```

We make wire references an instance of `Boolean` and `BooleanXZ`. The main helper function is `newItem`, which adds a node to the DAG and returns its reference. This is largely trivial, except for perhaps `zC` and `fromBoolean`, which are as follows (the rest of the instance is omitted):

```
instance BooleanXZ (WithDAG tp) (WireReference tp) where
  zC = newItem (MSymbolic CONNECT [])
instance Boolean (WithDAG tp) (WireReference tp) where
  fromBool b
    = newItem (MSymbolic (if b then AND else OR) [])
```

### A.3.2 Optimizations on symbolic instances

To work with symbolic instances, most techniques require the underlying DAG to be acyclic. We already guarantee this in our underlying data-structure. The procedure to substitute wires with their values, however, can re-introduce cycles. When done naively, this would cause infinite loops. Instead, we show how to perform this substitution, while keeping the resulting network cycle-free. We use the algorithm to eliminate cycles as illustrated in the previous chapter.

After parsing a module, we obtain a map describing a reference to a value (in the DAG) for each wire[2]. The `nmos` gate is symbolically expressed using a gated buffer, `GBUF` in our code. We combine this node with a unary `NOT`, to get a `pmos` gates. By using this combination, we only have to maintain the implementations for `GBUF`. Similarly, `cmos` gates use three DAG nodes: one for the `nmos` and two for the `pmos`. In some cases we introduce a node simply because it is used a lot in most designs. Such is the case with the `ITE` node. To give a bit of a flavour of what a DAG looks like internally, we list some nodes below. These nodes were generated after parsing our example design:

```
{- some DAG nodes:
 1: Symbol "dataLeft"
 2: Symbol "dataRight"
 4: Symbol "leftToRt"
 8: Symbol "rightToLt"
21: Symbolic BUFIF 20 2
```

[2]We do not consider the code to actually parse the Verilog.

```
22: Symbolic BUFIF 4 2
23: MSymbolic CONNECT [21,22]
42: MSymbolic CONNECT [25,26,21,22] -}
```

Some observations can be made about this DAG. First, very few optimizations are done. In fact, the only optimization done is grouping of associative operations. This can be seen in node `42`, which was constructed from node `23` and another node. In node `42`, the values of node `23` are repeated, such that node `23` is no longer referenced in node `42`, thus making node `23` obsolete. The obsolete node is not removed during this parse step, and duplicate nodes are not merged.

The map with wire-value references is only insightful in combination with the DAG. We look at three map elements: For wire `writeData`, the node referenced is `42`.

In addition to the DAG node, some wires also contain a reference to a module instance. For example, `writeData` has one to the instance `data_register`. This accounts for the possibility that that module writes data – later analysis will show it does not. At DAG node `4`, wire `leftToRt` occurs as a symbol. As a consequence, node `leftToRt` may be used in the nodes with a number higher than `4`. This introduces cycles, while keeping the `DAG` cycle-free. In this example, such a cycle exists in the value of `leftToRt`.

### A.3.3 Eliminate cyclic dependencies

Our strategy for eliminating cycles as presented in Section 3.4.1, comes from the observation that the value of a wire can be changed by a gate at most twice. Once from Z to 0 or 1, and then once to X. To remove cycles, we follow the nodes, while keeping track of which nodes have been visited. If every node has been visited, we refer to value X.

The code to follow a single node is called `incrementMapLoopless`. This is its code:

```
type State a
  =  ( a -- next unique value
     , IntMap.IntMap ( a -- new node number
                     , Symbolic a) -- its value
       -- top layer
     , [IntMap.IntMap ( a -- new node number
                      , Symbolic a) -- its value
       ] -- list of next layers
       -- empty list stands for infinite empty maps
     , Map.Map (Symbolic a) a -- reverse lookup
     )
incrementMapLoopless
  :: forall a. -- we can't mix up a and Int
     (Enum a, Ord a, Show a)
  => -- a function to obtain original values:
     ( Int -> Symbolic Int )
  -> IntSet.IntSet -- check cycles (initially empty set)
```

```
  -> IntSet.IntSet -- nodes which resulted in a new layer
  -> State a
  -> Int -- reference to follow
  -> ( State a
     , a -- a new reference (in the new map)
     )
incrementMapLoopless lookupf cycleCheck layerNodes st nr
 = if IntSet.member nr cycleCheck
   then addTop (workSharing incrWLayer (stripTop st))
   else workSharing incrWithinSameLayer st
  where
   -- recursive call with proper cycle-protection arguments
   symbol = if IntSet.member nr layerNodes then X
            else lookupf nr
   newCheck = IntSet.insert nr cycleCheck
   incrementWith :: IntSet.IntSet -> State a
                 -> (State a, Symbolic a)
   incrementWith useLayers useState
    = mapAccumL (incrementMapLoopless lookupf newCheck
                                      useLayers)
                useState symbol
   incrWLayer = incrementWith (IntSet.insert nr layerNodes)
   incrWithinSameLayer = incrementWith layerNodes
   -- moving to and from the next layer
   stripTop :: State a -> State a
   stripTop (nextA,_,layers,revmp)
    = case layers of
        []     -> (nextA,IntMap.empty,[],revmp)
        (h:tl) -> (nextA,h           ,tl,revmp)
   addTop :: (State a, a)
          -> (State a, a)
   addTop ((nA,              h,  tl,revL),v)
        = ((nA,currentTopLayer,h:tl,revL),v)
   currentTopLayer = let (_,tp,_,_) = st in tp
   -- check if any work needs to be done, share nodes
   workSharing doWork myState@(_,topLayer,_,_)
    = case IntMap.lookup nr topLayer of
       -- use the current node from this layer if it exists
       Just v -> ( myState, fst v )
       Nothing-> shareOrAdd (doWork myState)
   -- share a node if possible, but do not share X or Z
   shareOrAdd (newSt,X) = addNode (newSt,X)
   shareOrAdd (newSt,z@(MSymbolic CONNECT []))
    = addNode (newSt,z)
   shareOrAdd (newSt@(_,_,_,newRevMap), newSymbol)
    = case Map.lookup newSymbol newRevMap of
       Just v -> (newSt, v)
```

```
      Nothing -> addNode (newSt,newSymbol)
  -- create the node referenced as 'newNextA'
  addNode :: (State a, Symbolic a)
          -> (State a, a)
  addNode ( (newNextA,newTopLayer,newLayers,newRevMap)
          , newSymbol)
   = ( ( succ newNextA
       , IntMap.insert nr (newNextA,newSymbol) newTopLayer
       , newLayers
       , Map.insert newSymbol newNextA newRevMap
       )
     , newNextA)
```

Function `incrementMapLoopless` takes five arguments. The fourth argument is a structure that can be thought of as a 'state'. Its last argument is an `Int` that indicates which node should be followed. Its return type is the new state, together with a reference of type `a` to indicate the position where the followed node can be found in the new map. Note that the new map uses type `a` only to avoid confusion with the old map of type `Int`. In practice, `a` will be `Int`.

If the argument to be followed is in the `IntMap` of treated nodes, nothing is done: the state triple remains the same and we return the element found. If not, we insert a new element into the map of treated nodes, and increase the 'next unique value' (`nextA`) by one. Function `workSharing` does this. To get that new element, we may need to do recursive calls to `incrementMapLoopless`. In a recursive call, the previously visited nodes are kept in variable `cycleCheck`. When we encounter a node that is already visited, we jump to another layer. If we encounter a visited node, we return a node that stands for $\bot$. Value `X` is used as a value for `symbol`.

The type of `incrementMapLoopless` does not use our `WithDAG` protective wrapper. It is not intended for direct use. We export a function called `selfSimplify` that does exactly the type of simplification that is required for modules. We give its type, without the implementation:

```
selfSimplify :: forall ap x. Traversable x
             => (forall tp.
                 WithDAG tp ( Map.Map ScopedWireName
                                      (WireReference tp)
                            , x (WireReference tp)
                            ))
             -> WithDAG ap ( Map.Map ScopedWireName
                                      (WireReference ap)
                           , x (WireReference ap))
```

The map (in the first part of its argument) contains values for each wire. The second part gives a set of nodes that, for whatever reason, should also be in the new DAG (they will be optimized away otherwise). The type transformation from `tp` to `ap` ensures that we do not accidentally lose nodes by forgetting to pass them.

## A.4 Multiple analysis methods in a single tool

In our method, we allow pieces of the design to be translated as a black box. For real designs, this is a requirement for several reasons. First, our data-structure of DAGs only contains combinational gates. The smallest state-holding modules – usually registers and memory blocks – have to be treated as black box modules.

Another reason is that using a black box module can help perform verification on a design. In communication fabrics, queues are common state-holding elements. In addition, they are often easily identifiable. The verification methods in this thesis rely on annotated queues. We therefore treat queues as black box modules instead of merely treating the registers inside them as such.

Finally, some designs may contain lookup tables with 'arbitrary' routing information, or processor nodes that perform some 'arbitrary' function. To prove something about a network for all possible routing functions, or all possible processor nodes, we can treat these parts of the design as black box modules as well.

A black box Verilog module has wires connecting it to its environment. To reason about the interaction of the module with its environment, we use separate files for module annotations. An example syntax would be the following:

```
annotate flop;
  module reg_module_no_reset;
  value_in drive;
  value_out q;
  value_out !notQ;
  clock clk;
  set 1'b0;
  reset 1'b0;
endannotate
```

The intention of such annotation would be that verification methods that know about `flop`, will know what to use as driving value, and what to use as output value. The `annotate` syntax is parsed as a list of key-value pairs, in which multiple values per key are allowed. Keys are property names, and values are Verilog (right-hand-side) expressions. The only exception is the `module` property, which takes a module name. In our example, `value_out` appears twice to indicate that `q` and `!notQ` both act as output value. Methods unaware of clocks, sets or resets, simply ignore the fields `clock`, `set` or `reset`. Methods unaware of flops simply treat all modules with the name `reg_module_no_reset` as a black box. We also allow properties concerning multiple bits, which is useful for handling data wires in the case of queues. The endianness of such properties is determined by the declarations inside the mentioned module(s). Since the endianness of a wire influences how an expression is parsed, each expression must be parsed once for every module mentioned.

### A.4.1 Switching between different Boolean representations

Given a symbolic value, we might like to translate it into a different format. To do so, we implement a function to take one of the symbolic nodes, and return any

other instance of the Boolean class. This way, we can translate a module, simplify its values, and then perform different types of analysis. The central function for such translations is `symbolicToBoolean`:

```
symbolicToBoolean :: (BooleanXZ m b, Applicative m)
                  => (ScopedWireName -> m b) -> (a -> m b)
                  -> Symbolic a -> m b
symbolicToBoolean f myLookup = join . myTranslate
 where
  myTranslate (Symbol a) = return$ f a
  myTranslate (Symbolic opr a1 a2)
   = translateOpr opr <$> myLookup a1 <*> myLookup a2
  myTranslate (MSymbolic mop aLst)
   = translateMOpr mop <$> traverse myLookup aLst
  myTranslate (Unary op a)
   = translateUnary op <$> myLookup a
  myTranslate (ITE a b c)
   = iteC <$> myLookup a <*> myLookup b <*> myLookup c
  myTranslate X = return$ xC

translateUnary :: forall boolean m. BooleanXZ m boolean
               => UnaryOperator -> boolean -> m boolean
translateUnary NOT = notC
translateUnary BUF = bufC
```

Of helper functions `translateOpr`, `translateMOpr` and `translateUnary`, only the latter is presented (as it is the smallest of these three). Functions `translateOpr` and `translateMOpr` are similar.

One of the uses of `symbolicToBoolean`, is to display the nodes as a formula. To achieve this, `Bytestring` is made an instance of the class `Boolean` and `BooleanXZ`.

```
instance (Monad m, Applicative m)
    => BooleanXZ m ByteString where
 gbufC a b = return$ "gbuf("<>a<>","<>b<>")"
 combineC a b = return$ "combine("<>a<>","<>b<>")"
 bufC a = return$ "buf("<>a<>")"
 xC = return$ "X"
 zC = return$ "Z"
```

The `Boolean` instance of `Bytestring` is similar, and this is all there is to it to interface with a new internal data structure. We can use `symbolicToBoolean` with this new instance to see the wire values of the two-way repeater module from the beginning of this section. As the `Bytestring` data-structure cannot be used to perform the `selfSimplify` procedure (variable substitution would become very tricky), the `selfSimplify` procedure is used on the instance `Boolean (WithDAG tp) (WireReference tp)`. We disabled some additional simplification procedures before generating this output.

```
Module TwoWayRepeater
dataRight := combine(gbuf(!or(fromRight,!or(fromLeft,!Z)...
reset := Z ;
enabled := Z ;
fromRight := Z ;
dataLeft := combine(gbuf(!or(fromLeft,!or(fromRight,!or(...
fromLeft := Z ;
clock := Z ;
-----------------------
rightToLt := !or(fromRight,!or(fromLeft,!or(fromRight,!o...
outData := data_register>_q_0 ;
leftToRt := !or(fromLeft,!or(fromRight,!or(fromLeft,!Z))) ;
writeData := combine(gbuf(!or(fromRight,!or(fromLeft,!Z)...
wasntEnabled := !was_enabled_reg>reg>_q_0 ;
wasEnabled := was_enabled_reg>reg>_q_0 ;
readData := combine(gbuf(was_enabled_reg>reg>_q_0,data_r...
```

The symbolic output shows that both register modules were instantiated. The value for `outData` is given a 'fresh' variable `_q_0`, from the scope of `reg`, which is inside `data_register`. Apparently, `reg` is the name of the instantiation used inside the module `reg_module_no_reset`. The value of `leftToRt` is fully expressed using three nor gates. The input for which the circuit is not combinational, namely `fromRight = fromLeft = 0`, would yield X for `leftToRt`. This is the desired value. The value for `rightToLt` is slightly larger, as it reuses DAG nodes created for `leftToRt`. This shows that our strategy of reusing DAG nodes, while keeping the number of DAG nodes small, creates larger formulas than strictly necessary. In fact, removing cycles *does* cause an exponential blowup in the size of the final formula size, when compared to the smallest possible formula size. Such a blowup is common in many DAG to formula conversions.

In relation to black boxes that have been annotated with their properties, we can use the same translation:

```
transProps :: forall m b x.
              (BooleanXZ m b, CreateAtom ScopedWireName m b)
           => InverseProps (WireReference x)
           -> WithDAG x (InverseProps (m b))
transProps lst
 = flip map lst <$> maPure transFN
transFN :: forall m b t.
           (CreateAtom ScopedWireName m b, BooleanXZ m b)
        => Sequence.Seq (Symbolic Int) -- the DAG
        -> (WireReference t -> m b)
transFN mavec
 = (\(WireRef nr) -> findNr nr)
 where findNr = Sequence.index transVec
       transVec
        = map (symbolicToBoolean createAtom findNr) mavec
```

The black box module `reg_module_no_reset` is used once in the our ex-

ample. Using the `Bytestring` notation, we can see how these properties have been translated:

```
flop in data_register
  value_in := combine(gbuf(!or(fromRight,!or(fromLeft,!Z...
  value_out := data_register>_q_0;
  value_out := !data_register>notQ;;
  clock := clock;
  set := 0;
  reset := 0;
```

When creating designs, visualising formulas (in `Bytestring` notation) is quite helpful. When performing verification, other instances of `BooleanXZ` are used, but we can be sure that their input is equivalent to the formula produced in `Bytestring` notation, thanks to the class mechanism used.

### A.4.2 Boolean instance from Rings

We know that Booleans form a Ring. Here, we do the opposite: define a Ring based on a Monoid. We define our own classes, as the ones on 'hackage' are too fine-grained.

```
class Monoid m => Abelian m where
  (</>) :: m -> m -> m
  aTimes :: Int -> m -> m
class Abelian m => Ring m where
  rOne :: m
  (.*) :: m -> m -> m
```

In our interpretation, we use `rOne` for 'true', and `mempty` (from the `Monoid` class) for 'false'. We call our new instance `NaturalBoolean`, and do not assume that `rOne` squared is `mempty`, as is the case in Boolean rings. While this allows us to create 'non-standard' Boolean values (like `rOne` squared in case it is unequal to `mempty`) through the `Ring` structure, we cannot create these through the `Boolean` interface. By the laws that should hold in Rings, the only resulting values that can be obtained through the `Boolean` interface are (or ought to be equivalent to) `mempty` and `rOne`.

```
newtype NaturalBoolean m
  = NaturalBoolean {runNaturalBoolean :: m}
  deriving (Monoid,Abelian,Ring)

instance Ring m=> Boolean Identity (NaturalBoolean m) where
 fromBool b = return$ if b then rOne else mempty
 notC c     = return$ rOne </> c
 andC a b   = return$ a .* b
 orC a b    = return$ (a <> b) </> (a .* b)
 xorC a b   = return$ (a <> b) </> aTimes 2 (a .* b)
 iteC a b c = return$ (a .* b) <> (c .* (rOne </> a))
```

We can turn our 'Ring' into an instance of `BooleanXZ`, for which functions `andC` etc. stand for their Boolean (two-valued) counterparts, in the way we proposed in Chapter 3. In order to do so, we need to be able to control a single free Boolean variable. The class `SingleHelper` provides this.

```
class Show b => SingleHelper b where
  singleHelper :: b
  substituteHelper :: Bool -> b -> b
  helperDependent :: b -> Bool
  helperDependent _ = True -- default implementation
```

Function helperDependent is assumed to be an over-approximation. When it returns `False` on some term, that term with instances of the helper replaced by `True`, should be equivalent to the term with instances replaced by `False`. By allowing for over-approximations, faster tests like searching for occurrences of the helper, or simply always returning `True`, are fine as well. Our instance will be a new data type `DuoXZ` that allows us to distinguish between values that might be four-valued, and those that are guaranteed to be two-valued. Since the greatest portion of our design will use just the two-valued fragment, it pays off to make this distinction.

```
data DuoXZ b
 = DuoXZ {runDuoXZ :: b}
 | Duo {runDuoXZ :: b}
```

We assume that when `b` is used in `DuoXZ b`, it is an instance of `Boolean`, and of `SingleHelper`. This assumption is slightly stronger than the requirement that `m b` would be an instance of `SingleHelper`, but there have been no problems in defining suitable instances of `SingleHelper` so far. Here are the `Boolean` and `BooleanXZ` instances of `DuoXZ`:

```
instance (Boolean m b, SingleHelper b)
          => BooleanXZ m (DuoXZ b) where
 zC = return$ singleHelper  -- identity function
 xC = DuoXZ <$> notC singleHelper
 gbufC a b = DuoXZ <$> join (iteC (runDuoXZ a)
                                  <$> (runDuoXZ <$> (bufC b))
                                  <*> return singleHelper)
 bufC (DuoXZ b)
  = DuoXZ
    <$> join (iteC singleHelper <$> andC b0 b1
                                <*> orC  b0 b1)
   where b0 = substituteHelper False b
         b1 = substituteHelper True  b
 bufC b = return b
instance (Boolean m b, SingleHelper b)
   => Boolean m (DuoXZ b) where
 fromBool b = Duo <$> fromBool b
 notC (DuoXZ a) = bufC =<< (DuoXZ <$> notC a)
 notC (Duo b) = Duo <$> notC b
```

Again, some of the definitions are omitted for brevity. In particular, the defaults for the `Boolean` class do not apply, as they assume Boolean laws, instead of the truth tables from the Verilog standard, so all of these function had to be defined. To summarize, we can create an instance of `BooleanXZ` given an instance of `Ring` and `SingleHelper`. Chapter 4 shows how this can be useful.

# Glossary

**black box** A hardware module of which the implementation is either not available, or hidden on purpose. 9, 10, 23, 24, 81, 96, 114, 116

**communication fabric** A design that is used to connect components such that they can exchange information. 1, 3, 9, 12, 55, 56, 63, 75, 89, 101, 114

**CPU** Central Processor Unit. 1

**DAG** Directed Acyclic Graph. 35, 38, 91, 92

**eager** Always ready. For sinks: always ready to receive a packet. For sources: always ready to send one.. 66, 68, 69, 71, 73

**FPGA** Field Programmable Gate Array. 2

**gate level** Gate level netlist, a description of hardware based on logic gates. 1–6, 9–11, 24, 26, 28

**IP** Intellectual Property. 1, 2

**nMOS** n-type Metal-Oxide-Semiconductor field effect transistor. 39, 40

**NoC** Network On Chip: a communication fabric on an integrated circuit. 1, 3, 7, 9, 25, 95, 96

**NP** Class of problems that can be solved on a non-deterministic turing machine in polynomial time. A NP-hard problem is a problem to which each problems in this class can be reduced. An NP-complete problem is an NP-hard problem which is itself in the class NP. It is commonly assumed that NP-hard problems do not have polyonimal-time algorithms on conventional computers. 36, 90

**one-hot** Encoding of $n$ different values, using $n$ different wires of which only one can be high at any point in time. 91

**pMOS** p-type Metal-Oxide-Semiconductor field effect transistor. 39

**PSPACE** Class of problems that can be solved on a turing machine with polynomial memory. See also NP for the terms 'PSPACE-hard' and 'PSPACE-complete'. It is commonly assumed that there are strictly more problems in PSPACE, then there are in NP. 90, 92

**queue** A hardware module which can receive and store packets, hold a certain number of these, and send stored packets. 9

**RTL** Register Transfer Level: hardware abstraction level in which the next state for each registers is given.

**SAT** Satisfiability: a problem format that describes a quantifier free first order Boolean formula, for which a valuation (if it exists) can be found by a large number of tools, called SAT solvers. The stated problem asks: can a Boolean formula over a set of propositions can be satisfied. If so, the problem is called satisfiable, or SAT. If not, the problem is called UNSAT. 6, 26, 28, 39, 42, 43, 67, 68, 73, 83, 86, 90, 92, 96, 97

**SMT** Satisfiability Modulo Theories. A variant of SAT, in which propositions can be formulas over a certain theory. In this thesis, we use the theories of integers and real numbers. 55, 59, 60, 62, 63, 81, 82, 90, 96

**SoC** System on Chip. 1

**synthesis** The translation of a hardware description language (like Verilog) into a gate level language. 5

**trace** A (possibly infinite) sequence of states of a system, describing one possible way for the system to behave. 96, 97

**Voi** Our Haskell tool, available from `http://sjcjoosten.nl/voi/`. 23, 75, 80, 83, 96, 97, 99, 100, 108

**xMAS** Executable Micro-Architectural Specification, a language to specify networks. 7, 11, 15, 16, 20, 21, 23, 25, 45, 46, 55, 65–68, 70–73, 75, 76, 83, 86

# Bibliography


Aggarwal, P., Chu, D., Kadamby, V., and Singhal, V. (2011). Planning for end-to-end formal using simulation-based coverage: Invited tutorial. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 9–16, Austin, TX. FMCAD Inc. Cited on page 2.

Backes, J., Fett, B., and Riedel, M. D. (2008). The analysis of cyclic circuits with Boolean satisfiability. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 143–148. IEEE. Cited on page 43.

Backes, J. and Riedel, M. D. (2012). The synthesis of cyclic dependencies with boolean satisfiability. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(4):44. Cited on page 43.

Bergstra, J. A. and Ponse, A. (1999). Process algebra with five-valued conditions. Cited on page 25.

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without bdds. In Cleaveland, W., editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg. Cited on pages 55 and 59.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003). Bounded model checking. *Advances in computers*, 58:117–148. Cited on page 5.

Boule, M. and Zilic, Z. (2005). Incorporating efficient assertion checkers into hardware emulation. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 221–228. IEEE. Cited on page 2.

Bradley, A. R. (2011). Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer. Cited on page 95.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014). The nuxmv symbolic model checker. In *Computer Aided Verification*, pages 334–342. Springer. Cited on pages 86 and 97.

Chatterjee, S. and Kishinevsky, M. (2012). Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Formal Methods in System Design*, 40(2):147–169. Cited on pages 7, 15, 46, 76, and 86.

Chatterjee, S., Kishinevsky, M., and Ogras, Ü. Y. (2010). Quick formal modeling of communication fabrics to enable verification. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'10)*, pages 42–49. Cited on page 11.

Chatterjee, S., Kishinevsky, M., and Ogras, Ü. Y. (2012). xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Design & Test of Computers*, 29(3):80–88. Cited on pages 11, 15, 16, and 72.

Coffman, K. (1999). *Real world FPGA design with Verilog*. Pearson Education. Cited on page 104.

Drechsler, R., Eggersgluss, S., Fey, G., Glowatz, A., Hapke, F., Schloeffel, J., and Tille, D. (2008). On Acceleration of SAT-Based ATPG for Industrial Designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1329–1333. Cited on page 42.

Duato, J. (1993). A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 4:1320–1331. Cited on page 9.

Duato, J. (1995). A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067. Cited on page 9.

Edwards, S. A. (2003). Making cyclic circuits acyclic. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 159–162, New York, NY, USA. ACM. Cited on page 43.

Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer. Cited on page 83.

Eggersglüß, S., Fey, G., Glowatz, A., Hapke, F., Schloeffel, J., and Drechsler, R. (2010). MONSOON: SAT-based ATPG for path delay faults using multiple-valued logics. *Journal of Electronic Testing*, 26(3):307–322. Cited on page 43.

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. (2002). Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer. Cited on page 83.

Farahmandi, F. and Alizadeh, B. (2015). Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *Microprocessors and Microsystems*, 39(2):83 – 96. Cited on page 95.

Gao, M. and Cheng, K.-T. (2010). A case study of time-multiplexed assertion checking for post-silicon debugging. In *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, pages 90–96. Cited on page 2.

van Gastel, B., Verbeek, F., and Schmaltz, J. (2014). Inference of channel types in micro-architectural models of on-chip communication networks. In *Proceedings of the 22nd IFIP/IEEE International Conference on Very Large Scale Integration*. Cited on page 15.

Gebremichael, B., Vaandrager, F., Zhang, M., Goossens, K., Rijpkema, E., and Rădulescu, A. (2005). Deadlock prevention in the Æthereal protocol. *Correct Hardware Design and Verification Methods*, 3725/2005:345–348. Cited on page 9.

Goossens, K., Dielissen, J., and Radulescu, A. (2005). Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421. Cited on page 9.

Goossens, K. G. (1993). *Embedding hardware description languages in proof systems*. PhD thesis, University of Edinburgh. Cited on page 25.

Gotmanov, A., Chatterjee, S., and Kishinevsky, M. (2011). Verifying deadlock-freedom of communication fabrics. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, volume 6538, pages 214–231. Cited on pages 7 and 15.

Gupta, A. (1993). Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pages 5–92. Springer. Cited on page 2.

Gupta, A. and Selvidge, C. (2012). Acyclic modeling of combinational loops. US Patent 8,181,129. Cited on page 43.

Haynal, S., Kam, T., Kishinevsky, M., Shriver, E., and Wang, X. (2008). A system verilog rewriting system for rtl abstraction with pentium case study. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 79–88. Cited on page 5.

Hunt, W. J. and Swords, S. (2009). Centaur technology media unit verification. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, volume 5643, pages 353–367. Springer Berlin Heidelberg. Cited on pages 6, 43, and 80.

Hunt Jr, W. A. (1989). Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460. Cited on page 6.

IEEE (2001). IEEE Standard Verilog Hardware Description Language. Cited on pages 3 and 32.

IEEE (2009). IEEE Standard VHDL Language Reference Manual. Cited on page 3.

IntelPR (2011). Intel identifies chipset design error, implementing solution. `http://newsroom.intel.com/community/intel_newsroom/blog/2011/01/31/intel-identifies-chipset-design-error-implementing-solution`. Cited on page 2.

Joosten, S. and Joosten, S. J. C. (2015). Type checking by domain analysis in ampersand. In *RAMICS 2015, 15th International Conference on Relational and Algebraic Methods in Computer Science, Braga*. Cited on page 12.

Joosten, S. J. C., van Gastel, B., and Schmaltz, J. (2013). A macro for reusing abstract functions and theorems. In Gamboa, R. and Davis, J., editors, *International Workshop on the ACL2 Theorem Prover and its Applications*, volume EPTCS 114, pages 29–41. Cited on page 12.

Joosten, S. J. C., Kaliszyk, C., and Urban, J. (2014a). Initial experiments with TPTP-style automated theorem provers on ACL2 problems. In *International Workshop on the ACL2 Theorem Prover and its Applications*, volume 152, pages 77–85. Cited on page 12.

Joosten, S. J. C. and Schmaltz, J. (2013). Generation of inductive invariants from register transfer level designs of communication fabrics. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 57–64. IEEE. Cited on pages 10 and 11.

Joosten, S. J. C. and Schmaltz, J. (2014). Scalable liveness verification for communication fabrics. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 113:1–113:6. Cited on page 11.

Joosten, S. J. C. and Schmaltz, J. (2015). Automatic extraction of micro-architectural models of communication fabrics from register transfer level designs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 1413–1418. Cited on page 11.

Joosten, S. J. C., Verbeek, F., and Schmaltz, J. (2014b). WickedXmas: Designing and verifying on-chip communication fabrics. In *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*. Cited on page 12.

Joosten, S. J. C. and Zantema, H. (2013). Relaxation of 3-partition instances. In *CTW*, pages 133–136. Cited on page 12.

Kaufmann, M., Moore, J., Davis, J., and 'numerous members of the ACL2 community' (2015). ACL2 XDOC manual: `4v-<=`. `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2____4V-_C3_D3`. Cited on page 42.

Kim, N., Park, J., Singh, H., and Singhal, V. (2014). Sign-off with bounded formal verification proofs. In *Design and Verification Conference and Exhibition*. Cited on page 2.

Kovásznai, G., Veith, H., Fröhlich, A., and Biere, A. (2014). On the complexity of symbolic verification and decision problems in bit-vector logic. In *Mathematical Foundations of Computer Science 2014*, pages 481–492. Springer. Cited on page 93.

Kropf, T. (2013). *Introduction to formal hardware verification*. Springer Science & Business Media. Cited on page 2.

Kuehlmann, A. (2004). Dynamic transition relation simplification for bounded property checking. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 50–57. Cited on page 6.

van Laarhoven, T. (2012). Dependently typed dags. `http://www.twanvl.nl/blog/haskell/dependently-typed-dags`. Cited on page 107.

Launchbury, J. and Peyton Jones, S. L. (1994). Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM. Cited on page 106.

Melham, T. F. (1990). Abstraction mechanisms for hardware verification. In Yoeli, M., editor, *Formal Verification of Hardware Design*, pages 30–49. IEEE Computer Society Press. Cited on page 3.

Mills, D. (2012). Yet another latch and gotchas paper. In *Synopsys Users Group (SNUG) Sylicon Valley*. Cited on page 42.

Minato, S.-I. and Somenzi, F. (1997). Arithmetic boolean expression manipulator using bdds. *Formal Methods in System Design*, 10(2-3):221–242. Cited on page 61.

Mishchenko, A., Chatterjee, S., Jiang, R., and Brayton, R. K. (2005). Fraigs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report. Cited on page 92.

Neiroukh, O., Edwards, S., Song, X., et al. (2008). Transforming cyclic circuits into acyclic equivalents. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1775–1787. Cited on page 43.

Ray, S. (2013). *Scalable Model Checking Beyond Safety-A Communication Fabric Perspective*. PhD thesis, University of California, Berkeley. Cited on pages 3 and 7.

Ray, S. and Brayton, R. K. (2012). Scalable progress verification in credit-based flow-control systems. In *DATE*, pages 905–910. Cited on pages 3, 7, 15, 45, 72, 75, 76, and 78.

Riedel, M. D. and Bruck, J. (2007). Synthesis of cyclic combinational circuits. US Patent 7,249,341. Cited on page 43.

Riedel, M. D. and Bruck, J. (2012). Cyclic Boolean circuits. *Discrete Applied Mathematics*, 160(13 - 14):1877 – 1900. Cited on pages 25, 26, 35, 37, and 43.

Rivest, R. L. (1977). The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers*, C-26(6):606–607. Cited on page 26.

Sutherland, S. and Mills, D. (2007). *Verilog and SystemVerilog Gotchas*. Springer US. Cited on page 42.

Synthesis, B. L. and Group, V. (2015). Abc: A system for sequential synthesis and verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`. Cited on page 97.

Turpin, M. (2003). The dangers of living with an X (bugs hidden in your verilog. In *Synopsys Users Group Meeting*. Cited on page 42.

Venu, B. and Singh, A. (2012). Formal verification methodology considerations for network on chips. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 220–225, New York, NY, USA. ACM. Cited on page 9.

Verbeek, F., Joosten, S. J. C., and Schmaltz, J. (2013). Formal deadlock verification for click circuits. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 183–190. IEEE. Cited on page 12.

Verbeek, F. and Schmaltz, J. (2011a). A comment on "a necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks". *IEEE Transactions on Parallel & Distributed Systems*, 22(10):1775–1776. Cited on page 9.

Verbeek, F. and Schmaltz, J. (2011b). Hunting deadlocks efficiently in microarchitectural models of communication fabrics. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 223–231, Austin, TX. Cited on pages 15 and 86.

Verbeek, F. and Schmaltz, J. (2012a). Easy formal specification and validation of unbounded networks-on-chips architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(1):1:1–1:28. Cited on page 7.

Verbeek, F. and Schmaltz, J. (2012b). Towards the formal verification of cache coherency at the architectural level. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(3):20:1–20:16. Cited on page 7.

Wouda, S., Joosten, S. J. C., and Schmaltz, J. (2015). Process algebra semantics & reachability analysis for micro-architectural models of communication fabrics. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 Thirteenth ACM/IEEE International Conference on*. Cited on page 12.

Zhu, Q., Kitchen, N., Kuehlmann, A., and Sangiovanni-Vincentelli, A. (2006). Sat sweeping with local observability don't-cares. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 229–234. Cited on page 6.

# Summary

## Verification of interconnects

A communication fabric, or a Network on Chip (NoC), is a way to combine hardware components. For the final hardware to function correctly, it is critical that at least this NoC functions correctly. Some methods to prove the correctness of NoCs exist. An important property of correct NoCs, which turns out to be particularly difficult to prove, is that local deadlocks should not arise. We call NoCs live if they do not have such local deadlocks. Liveness proofs for networks that are large enough to be interesting in practice, used to exist only on an abstract level, or rely heavily on information from the abstract level.

To prove correctness of the concrete implementation of a NoC, we view the network as a set of queues an other state-holding elements, together with some combinational logic. This view lies very close to the actual hardware implementation. Queues and other state-holding elements are annotated through their interface. For this reason, we call this view on hardware the 'interface level'.

The gate level implementation of a NoC design can be translated to combinational logic. At the gate level implementations, hardware designers make use of open wires and multi-directional gates. Where usual translations of gate level implementations are restricted to acyclic circuits with only binary gates, we show how to translate a richer class of gates to Boolean formulas. These Boolean formulas represent the gate level implementation of a NoC design.

In a proof about the correctness of a NoC, it can be important to know whether some queues can hold packets in a specific way. Linear one-step inductive invariants turn out to be a powerful tool to answer this question. Finding the right linear one-step inductive invariant used to require a high level description of the NoC. The essential property that is used describes when a packet of a certain type enters or leaves a queue: when a transfer occurs. By translating this property to a linear term, we can identify the linear invariants about the NoC. This property turns out to be available at the interface level. We describe how to translate this property to a linear term. As a consequence, we can automatically derive linear inductive invariants at the interface level.

To determine whether or not a NoC is live turns out to be very challenging. At the abstract level, there is a method that answer this question only partially: given a NoC, it can either prove that it is live, or it will present a situation which might be a reachable local deadlock. The latter case leaves the option open that

the situation is not a reachable deadlock, in which case we still do not know whether or not the NoC is live. However, this methods has a good performance, and can determine liveness of NoCs of realistic sizes. We show that there is a similar algorithm at the interface level. Given a gate-level hardware description of a NoC, together with an interface specification of the queues, we formulate a Satisfiability Modulo Theory (SMT) problem that has an answer if the NoC has a reachable local deadlock. For many NoCs, the SMT problem turns out not to have an answer, which can be verified with standard SMT solvers. This proves that these NoCs are live, using only information that is available at the interface level.

We also reproduce an abstract-level description from the interface level. To do so, we give a procedure that constructs a tree of synchronising and arbitrating elements from the interface descriptions. Next, we orient the elements, deriving a graph in which all components are similar to the components in the xMAS language, a language for describing NoCs at an abstract level, proposed around 2010. For NoCs that were generated from xMAS, the resulting reproduced NoC is not necessarily equal to the original xMAS. Instead, we define a property called 'transfer equivalence', and show that the resulting NoCs are transfer equivalent.

In addition to describing these novel techniques to analyse NoCs at the interface level, we have implemented the techniques as a proof of concept. Most of those implementations are now also available as part of a tool called Voi, which stands for 'Verify on interfaces'. The tool Voi is implemented in Haskell, and we give some insight into its implementation. With this tool, we believe to have made a significant step towards the automated verification of gate-level descriptions of NoCs.

# Samenvatting

## Verificatie van interconnects

Een communicatienetwerk, of een Netwerk Op een Chip (NoC), is een manier om hardwarecomponenten samen te voegen. Om ervoor te zorgen dat de samengestelde hardware goed werkt, is het belangrijk dat de NoC dat in ieder geval doet. Er bestaan methoden om de correcte werking van NoCs vast te stellen. Een belangrijke eigenschap van correct werkende NoCs, die ingewikkeld is om te bewijzen, is dat er geen lokale deadlocks voor mogen komen. We noemen NoCs levend wanneer ze zulke lokale deadlocks niet hebben. Voor het levend zijn van netwerken die groot genoeg zijn om in de praktijk relevant te zijn, bestonden voorheen alleen bewijzen op een abstract niveau, of zij steunden sterk op informatie vanuit het abstracte niveau.

Om correctheid te bewijzen van een concrete implementatie van een NoC, beschouwen we het netwerk als een verzameling wachtrijen en andere elementen met een toestand, samen met wat logica. Deze manier van beschouwen ligt erg dicht bij de uiteindelijke hardwareimplementatie. Wachtrijen en andere elementen met een toestand worden aangeduid door hun interface. Om deze reden noemen we onze manier van het beschouwen van hardware het 'interfaceniveau'.

De implementatie van een NoC met logische-poorten, kan vertaald worden naar logica. Bepaalde logische-poorten worden door hardware ontwerpers gebruik om tot onverbonden draden en communicatie in meerdere richtingen te komen. Terwijl de meeste vertalingen van logische-poorten het verbieden dat er zulke constructies bestaan, laten wij zien hoe we een rijke klasse aan ontwerpen naar Boolse formules kunnen vertalen. Deze Boolse formules vertegenwoordigen het logische-poort-ontwerp van een NoC.

In een bewijs over de correctheid van een NoC kan het van belang zijn om te weten of wachtrijen bepaalde pakketten kunnen bevatten. Lineaire inductief-bewijsbare invarianten blijken een krachtige manier te zijn om die vraag te beantwoorden. Om de juiste lineaire inductief-bewijsbare invarianten te vinden, had men voorheen een abstracte representatie van het NoC nodig. De essentiële eigenschap die daarin gebruikt wordt, beschrijft wanneer een pakket van een bepaald type een wachtrij in of uit gaat: wanneer er een verplaatsing is. Door deze eigenschap als lineaire term te vertalen, kunnen we de lineaire invarianten van het NoC vinden. Deze eigenschap blijkt ook beschikbaar te zijn op het interfaceniveau. We beschrijven hoe we deze eigenschap moeten vertalen naar een lineaire term. Als

gevolg hiervan kunnen we automatisch de lineaire inductieve invarianten op het interfaceniveau vinden.

Besluiten of een NoC levend is, blijkt een uitdaging. Op het abstracte niveau is er een methode die dit gedeeltelijk beantwoord: gegeven een NoC, kan het bewijzen dat het levend is, of geeft het een situatie die een bereikbare lokale deadlock kan zijn. In het laatste geval blijft er de mogelijkheid bestaan dat die situatie geen bereikbare lokale deadlock is, waarna we nog steeds niet weten of de NoC levend is. Echter, deze methode werkt vlot, en kan van NoCs met een realistische grote vaststellen dat ze levend zijn. We laten zien dat er een vergelijkbaar algoritme op het interfaceniveau bestaat. Gegeven een logische-poort-ontwerp van een NoC, samen met een aanduiding van wachtrijen door middel van hun interface, kunnen we een vervulbaar-modulo-theorie (SMT) probleem opstellen, dat een antwoord heeft als het NoC een bereikbare lokale deadlock heeft. Voor veel NoCs heeft het SMT probleem geen antwoord, hetgeen automatisch kan worden vastgesteld door software voor SMT problemen. Dit bewijst dat de NoCs levend zijn, terwijl we alleen gebruik maken van informatie op het interfaceniveau.

We kunnen ook een abstracte representatie van het NoC geven vanuit het interfaceniveau. Om dat te doen, geven we een procedure die een boomstructuur van synchroniserende en bemiddelende elementen bouwt vanuit de beschrijvingen op interfaceniveau. Vervolgens geven we de elementen een richting, waaruit een graaf ontstaat waarin alle componenten lijken op componenten uit de taal xMAS, een taal om NoCs op een abstract niveau te beschrijven, die rond 2010 voorgesteld is. Voor NoCs die vanuit xMAS gegenereerd zijn, blijkt de resulterende NoC niet persé overeen te komen met de oorspronkelijke xMAS. We definiëren in plaats daarvan een eigenschap genaamd ‘verplaatsingsequivalent’, en laten zien dat de resulterende NoCs verplaatsingsequivalent zijn.

Naast dat we deze nieuwe NoC-analysetechnieken op het interfaceniveau beschrijven, hebben we de technieken ook geimplementeerd. De meeste van deze technieken zijn ook beschikbaar als onderdeel van software met de naam Voi, hetgeen staat voor ‘Verifieer op interfaces’. Voi is geschreven in Haskell, en we lichten enkele onderdelen van de implementatie toe. We verwachten met deze software een flinke stap gezet te hebben om logische-poort-ontwerpen van NoCs automatisch te verifiëren.

# Curriculum Vitae

Sebastiaan Joosten
12 January 1985
born in Enschede

## Education

**Stichting Facta** Apeldoorn
MG1, MG3 (MBO-level degrees in Computer Science), 1995

**Het Stedelijk Lyceum** Enschede
Gymnasium NT and NG with Latin and Computer Science, 2002

**University of Twente** Enschede
Propaedeutics Applied Physics, 2004
BSc Applied Physics, 2009
MSc Applied Mathematics, 2011 *cum laude*

**Utrecht Summer School** Utrecht
International School on Rewriting 2010, Advanced track

## Work experience

**Teaching assistant** University of Twente, 2004 - 2005

**Tutor VWO** Twente Academy, University of Twente, 2007, 2008

**Programmer**
Haskell, PHP, ADL, Open University of the Netherlands (OUNL), 2008, 2009
PHP, Flash, JavaScript, Axis Media Ontwerpers, Enschede, 2000 - 2010
PHP, Flash, Axis Media Ontwerpers, Berlin, 2002 - 2003

**PhD student** Effective Layered Verification of Networks on chips.
Open University (2012 - 2014), Technical University Eindhoven (2014 - 2015)

## Titles in the IPA Dissertation Series since 2013

**H. Beohar**. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

**G. Igna**. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

**E. Zambon**. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

**B. Lijnse**. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

**G.T. de Koning Gans**. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

**M.S. Greiler**. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

**L.E. Mamane**. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

**M.M.H.P. van den Heuvel**. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

**J. Businge**. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

**S. van der Burg**. *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

**J.J.A. Keiren**. *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11

**D.H.P. Gerrits**. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12

**M. Timmer**. *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

**M.J.M. Roeloffzen**. *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14

**L. Lensink**. *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15

**C. Tankink**. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants.* Faculty of Science, Mathematics and Computer Science, RU. 2013-16

**C. de Gouw**. *Combining Monitoring with Run-time Assertion Checking.* Faculty of Mathematics and Natural Sciences, UL. 2013-17

**J. van den Bos**. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01

**D. Hadziosmanovic**. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

**A.J.P. Jeckmans**. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

**C.-P. Bezemer**. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

**T.M. Ngo**. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

**A.W. Laarman**. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

**J. Winter**. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

**W. Meulemans**. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

**A.F.E. Belinfante**. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

**A.P. van der Meer**. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

**B.N. Vasilescu**. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

**F.D. Aarts**. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

**N. Noroozi**. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

**M. Helvensteijn**. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

**P. Vullers**. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

**F.W. Takes**. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

**M.P. Schraagen**. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

**G. Alpár**. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

**A.J. van der Ploeg**. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

**R.J.M. Theunissen**. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

**T.V. Bui**. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of

Mathematics and Computer Science, TU/e. 2015-04

**A. Guzzi**. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

**T. Espinha**. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

**S. Dietzel**. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

**E. Costante**. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

**S. Cranen**. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

**R. Verdult**. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

**J.E.J. de Ruiter**. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

**Y. Dajsuren**. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

**J. Bransen**. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

**S. Picek**. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

**C. Chen**. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

**S. te Brinke**. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

**R.W.J. Kersten**. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

**J.C. Rot**. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

**M. Stolikj**. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

**D. Gebler**. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

**M. Zaharieva-Stojanovski**. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

**R.J. Krebbers**. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22

**R. van Vliet**. *DNA Expressions – A Formal Notation for DNA.* Faculty of Mathematics and Natural Sciences, UL. 2015-23

**S.-S.T.Q. Jongmans**. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

**S.J.C. Joosten**. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02