Formal Verification of On-Chip Communication Fabrics

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen,
op gezag van de Rector Magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op
dinsdag 26 maart 2013 om 13:30 uur precies

door
Freek Verbeek
geboren op 17 september 1983
te Nijmegen

Promotoren:

prof. dr. F.W. Vaandrager
prof. dr. M.C.J.D. van Eekelen

Copromotor:

dr. J. Schmaltz

Samenstelling manuscriptcommissie:

prof. dr. H. Geuvers
prof. dr. N. Bagherzadeh (University of California, Irvine, CA)
prof. dr. D. Borrione (University Joseph Fourier, FR)
prof. dr. A. Jantsch (KTH Royal Institute of Technology, SW)
dr. M. Kishinevsky (Intel Corporation, OR)

Cover: Pim Sebok

This research is supported by the Netherlands Organisation for Scientific Research (NWO) under the project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811, and is supported by a grant from Intel Corporation.

ISBN 978-94-61083-91-3

Dankwoord

Er zijn ontzettend veel mensen die de afgelopen vier jaar in meer of mindere mate aan de totstandkoming van dit proefschrift hebben bijgedragen. Ik wil deze mensen hiervoor bedanken en een aantal er hier uitlichten.

Mijn copromotor, Julien, is essentieel geweest bij iedere stap die ik de afgelopen vier jaar heb gezet. Vier jaar lang hebben we samen hard gewerkt, vele discussies gehad, geluncht, aan conferenties deelgenomen, universiteiten en bedrijven verspreid over Europa en de VS bezocht, nieuwe contacten gelegd en af en toe een biertje gedronken. Ook al kunnen we een diepgaand meningsverschil hebben over de opzet van een intro – en ook al moest ik vaak een paper tot treurens toe "meer structuur" geven – in die vier jaar is er nooit een frustratie ontstaan en heb ik de samenwerking altijd als bijzonder prettig ervaren. Ik ben blij dat we ook in de toekomst nog samen zullen werken. Merci.

Frits en Marko, mijn promotoren, wil ik bedanken voor het vertrouwen en de vrijheid die ik de afgelopen vier jaar heb genoten. Enerzijds hebben jullie me volledig mijn eigen gang laten gaan, maar tegelijkertijd waren jullie altijd betrokken en hebben jullie constructief, gedetailleerd en bovenal nuttige feedback gegeven op mijn proefschrift.

Many thanks go to my manuscript committee, who did not only take the considerable effort of reading and critiquing my thesis to the last detail, but also of traveling many miles to be present at my defense. Professor Borrione, professor Jantsch and dr. Kishinevsky, I hope the future will hold many more collaborations. Professor Geuvers, Herman, to you I am greatly indebted for supervising my master thesis, suggesting me as a PhD candidate to Frits, and finally for presiding at my defense. A special thanks to professor Bagherzadeh and Abdulaziz Alhussien, whose inspiring ideas have had a great influence on this thesis and whose generous hospitality has left a great impression upon me.

Er is een hele lijst aan collega's om te bedanken, te veel om hier op te sommen. Mijn kamergenoten Bas, Faranak, Maarten, Martijn en Thomas hebben het promoveren af en toe stukken leuker gemaakt dan het eigenlijk is (de USB gestuurde rocket launcher heb ik altijd een goede investering gevonden). De mensen van MBSD, maar ook de rest van ICIS, wil ik bedanken voor de lunches, koffiepauzes, leesclubjes en borrels. Bas, Bernard en Tom, het was geweldig om met jullie samen te werken.

Het promoveren had ik hoogstwaarschijnlijk nog geen twee maanden volgehouden zonder alle vrienden om mij heen. Ik hoop dat iedereen zich realiseert dat ik ze hiervoor ontzettend dankbaar ben. In het bijzonder verdient Pim hier wat lovende woorden voor zijn prachtige design van de voorkant van dit proefschrift. Ik vind het fantastisch dat je niet alleen uitgebreid hebt geluisterd naar mijn uitleg over mijn promotie onderwerp, maar er ook nog genoeg van oppikte om een mooie deadlock in het ontwerp te verwerken.

Arthur en Fabian, paranimfen, het is voor mij een eer naast jullie te staan tijdens de verdediging [Mulders07]. En ook daarbuiten, natuurlijk.

Een promotietraject is nogal een aparte tijd. Een tijd vol onzekerheid over wat je moet doen en of dat gaat lukken, een tijd met veel deadlines en stress, een tijd waarin veel nieuwe en spannende ervaringen tegelijkertijd plaatsvinden. In zo'n tijd is de kennis dat er een constante factor is – namelijk dat er een plek is waar je altijd onvoorwaardelijk terecht kan – van onschatbare waarde. Lia en Jos, ik kan jullie niet genoeg bedanken voor alles wat jullie hebben gedaan.

Nijmegen, januari 2013

Contents

Part I

Preamble

Chapter 1
Introduction

As technology advances, microchips become more and more complex. To grasp the complexity of a future chip, consider the streets of a large city from an aerial view. Buildings – like black boxes – continuously inject and consume cars from the infrastructure, helicopters fly around and underneath it all a subway transports masses of people between fixed points. The integral behavior is dazzlingly complex and chaotic. Future microchips may well achieve a similar degree of complexity. On a single chip, hundreds of cores (the buildings) are performing parallel computations. A future on-chip infrastructure can integrate channels (the streets), wireless transmitters (the helicopters), or optical interconnects (the subways) on a few square centimeters. Even three-dimensional chips are feasible, essentially stacking several mutually connected cities upon each other.

The cause of all this on-chip complexity, is the ability of modern manufacturers to integrate huge amounts of transistors on a small surface. Currently, chips are built from transistors being only 22 nanometers long. Smaller transistors are expected (latest research has even developed a transistor consisting of only a single atom [57]) allowing more on-chip possibilities. It is up to both industry and academia to exploit this nano technology and to cope with the complexity inherent to running more than a billion transistors in one integrated circuit.

A general approach is to reduce the overall complexity by raising the level of abstraction of the design phase. Instead of building a chip from scratch, designers use prefabricated building blocks called intellectual properties or cores. A System-on-Chip (SoC) integrates different cores on one die. Examples of cores are processors, memory controllers, and video processing units. Cores are created, verified and tested individually, allowing a divide-and-conquer approach in designing complex microchips.

Essential to the performance of the chip is an efficient communication interconnect between these cores. Traditionally, a bus is used to facilitate all on-chip communications. The major drawback of a bus is scalability. It is possible to connect dozens of cores on one bus, but not hundreds. To deal with this issue, Networks-on-Chip (NoC) have been proposed [8]. In an on-chip network, several cores communicate with each other by transmitting messages which move through the interconnect until they arrive at their destination. An NoC is a “city on chip” with packets of data as its inhabitants.

The NoC paradigm introduces new challenges to the world of SoC designers. In a chip with a bus, it is relatively easy to ensure that a message sent from one core to another will always eventually arrive. In an on-chip network this becomes more troublesome. In the interconnect of an NoC scenarios may occur in which messages never arrive at their destination. A deadlock is such a situation. Figure 1.1 shows an example. The cubes represent cores that make use of an on-chip interconnect to transmit and receive messages. The on-chip infrastructure is depicted as a set of streets, and the messages moving through this interconnect are depicted as cars. Each message has a destination, and it knows which direction to take to get towards its destination. However, each message is waiting for another message. All messages are waiting permanently.

A deadlock is not the only scenario in which messages never arrive at their destination. Various network-related issues can occur such as misrouting, livelocks, and starvation. None of these issues may occur, if a network is to be called *correct*.

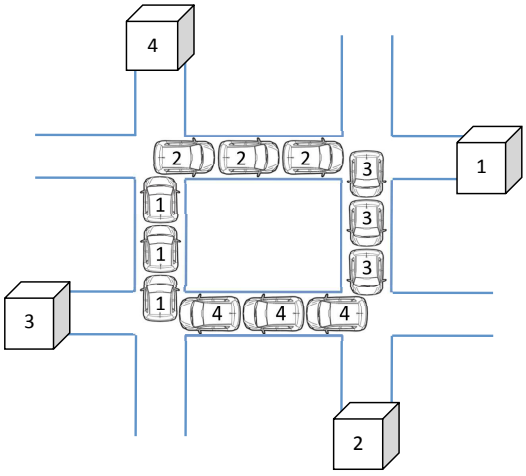


Figure 1.1: *Deadlock Scenario*

Complex as microchips may be, we base our everyday lives on the assumption that these chips correctly do what we expect them to do. Microchips are omnipresent, ranging from our laptops, to home appliances, to safety-critical systems. A chip that performs correctly often remains unnoticed. It is when bugs occur that we realize how consequential it is to assume correctness of all the hugely complex chips surrounding us. For example, in 2011 Intel identified a problem with its Cougar Point chipset. They had to halt shipments and repair existing systems, with a total cost of about 700 million dollars.

Establishing chip correctness – beyond any doubt – is a hard problem. With cores being validated separately, this thesis focuses on verification of the interconnect between the cores. A major difficulty is that correctness of the interconnect is an emergent property depending on many different facets of the NoC, such as which cores run on top of it, which routes can be taken by messages in the interconnect, and under which conditions messages are stalled.

Formal verification is a technique for assessing that a system is correct, i.e., that it meets a specification formalized in some mathematical language. Generally, a formal model of the system is created using, for example, state machines or Petri nets. A specification is formulated in a logic such as Linear Temporal Logic (LTL). It is then proven that the model always satisfies the specification.

Whether it concerns the formulation of definitions, the implementation of an algorithm, or the proof of a theorem, humans' labor is always error-prone. Wolper asserts that "manual verification [of a program] is at least as likely to be wrong as the program itself" [146]. A proof can only reliably be called sound if it is built from very small logical reasoning steps. However, even in the proofs of the smallest theorems humans have the tendency to think in larger steps, eschewing the rigority enforced by logic. The result is that human proofs are often unreliable. In contrast, the rigorous and fine-grained nature of logic perfectly suits a computer. The soundness of logical proofs can be verified more reliably by computers than by humans. It is therefore common practice to perform formal verification mechanically, that is, in such a way that its soundness can be ascertained by a computer.

Applying formal methods to on-chip interconnects provides trustworthy claims on their correctness. However, mechanical verification generally has to deal with a trade-off between ease of use and scalability. Automated and general techniques such as model checking are push-button solutions, but do not scale to the complexity of realistic NoCs. On the other hand, interactive and parametric techniques such as theorem proving are laborious and hard to use, which tends to prevent their widespread adoption. Currently, there is no way of formally verifying interconnects on the scale demanded by the NoC paradigm.

The contribution of this thesis consists of *easy* and *scalable* mechanical verification methods for communication networks. We formalize a notion of correctness, stating that a network is always eventually able to inject messages and that any injected message will always eventually arrive at its destination. We show that in order to establish this emergent correctness property for some NoC, it suffices to prove several smaller properties on isolated constituents. With a realistic example we conclude that many of these properties can easily be verified. Deadlock freedom, however, remains difficult to prove due to the interactions between the different constituents of the network. Therefore we provide formally proven correct tools and algorithms to hunt for deadlocks in communication networks.

This thesis advances the state-of-the-art both theoretically and practically. We provide new theories identifying deadlocks in communication networks. We prove that deciding deadlock freedom of wormhole networks is co-NP-complete. On the practical side, we design deadlock detection algorithms. A new tool is presented, which runs optimized C implementations of these algorithms in parallel. We present examples that could previously not be proven deadlock-free, due to either scalability issues or the complexity of dealing with many different facets of NoCs all at once. Our algorithms are able to either prove absence of deadlocks or report a counterexample. The correctness of our theories and algorithms has been established mechanically using the ACL2 theorem prover.

Before discussing the outline of this thesis, we present a short introduction into formal verification and communication networks. We then motivate the structure of this thesis using two examples. More background information and related work can be found in the next chapter.

1.1 Formal Verification

The current state-of-the-art in verifying on-chip networks is simulation [17, 15, 91]. Simulation can be used to predict the behavior of complex and interactive systems. An inherent disadvantage of simulation is the difficulty to obtain full coverage. Corner cases are hard to find and debug. Finding simulation patterns that cover them all is even more intricate. Simulation does not scale to future communication-centric SoCs [121]. The focus of this thesis is therefore on *analytical* formal verification approaches to complement simulation.

Analytical approaches do not consider dynamic and runtime behavior of a system, but statically analyze a system. Various different flavors of analytical mechanical verification exist. *Model checking* is an automated technique to check whether some model of a system satisfies a certain specification [81, 5]. The model is described in some sort of state machine and the specification is described in a temporal logic. A model checking algorithm uses the transition function associated to the state machine to explore the state space and to find states that do not satisfy the specification. If it finds such a state, both the state and the trace leading to this state are reported. If such a state is not found, the system is proven correct. Model checking is widely adopted by academia and industry alike, mostly because it is completely automatic and it can provide counterexamples. The major issue is a combinatorial blow-up of the number of states that needs to be explored, referred to as state space explosion. This severely limits the scalability of model checking.

Theorem proving is a technique where the proof of some mathematical theorem is formalized in such a way that a computer program can ensure its correctness. Theorem proving is an interactive process, during which the user supplies hints and lemma's to the theorem prover until the proof can be derived by the system. A theorem prover can guide the user while making his proof, by breaking down large proofs into smaller ones, by simplifying and rewriting the current goal, or by automatically searching for lemma's that can be used to prove the current theorem. The major advantage of theorem proving is the ability to deal with parametric systems. For example, the size of a network can remain unspecified, meaning that the theorem is proven to hold for networks with any size. Scalability is then measured by the amount of interaction with the user. Theorem proving is generally considered an academic effort, even though it has been adopted by various industry such as Intel and AMD. The process of theorem proving requires a relatively steep learning curve, and even for an experienced user it still often requires a great amount of interaction to prove non-trivial theorems.

SAT solvers are automated algorithms that decide whether some formula is satisfiable [100, 62]. Even though the satisfiability problem is well-known to be NP-complete, SAT solvers can prove truthness of propositional formulae with millions of variables. Satisfiability Modulo Theories (SMT) adds a background theory to a SAT solver, effectively giving an interpretation to some symbols [6]. For example,

the integer arithmetic theory assigns the expected interpretation to symbols such as $<$, $+$, and 0. Other examples of theories deal with various data structures such as lists, arrays, or bit vectors. The use of SAT solvers is completely automatic, but it requires the problem or theory that is to be proven to be formulated as a SAT instance. Many model checkers and theorem provers make use of SAT and SMT solvers under the hood.

This thesis applies both theorem proving and SMT solvers to perform parametric and scalable verification of NoCs. The ultimate purpose of this effort is to remove the traditional objection against theorem proving – namely that it is a difficult and time consuming process – while preserving the major advantage of scalable and analytical verification.

1.2 Communication Networks

In this thesis, the term communication network denotes a synchronous message passing network with a static topology. The behavior of all the switches in the network is predetermined and is assumed to be reliable. There is no packet loss, and communication wires do not cause any bit loss.

It is common practice to reason about the total functionality of a network in terms of different layers of abstraction defined by the OSI model [139]. This approach has also been adopted for on-chip networks [130, 88, 48]. Figure 1.2 shows the three different layers used in this thesis.

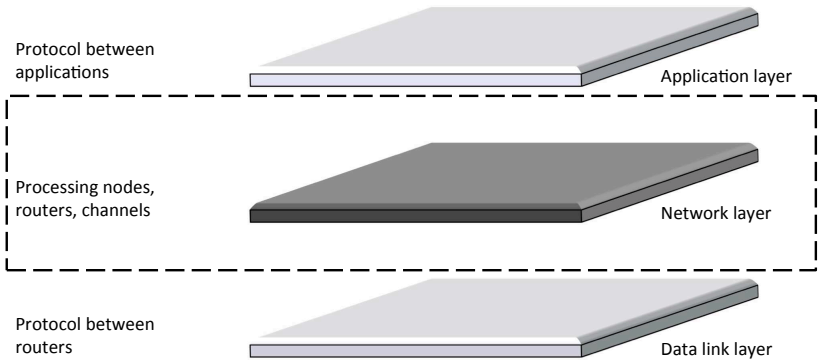


Figure 1.2: *Three layers occurring in this thesis.*

Application Layer

The highest layer considered in this thesis is the application layer. At this layer, applications (or cores) such as processors or memory controllers run in parallel. These cores communicate with other cores by transmitting messages back and forth

over the network. The behavior of the cores dictates where, when, and which types of messages are injected into the network. It also dictates where, when, and which types of messages are consumed from the network. The behavior of the cores will also be called the *application-layer protocol*.

For example, a cache coherency protocol between the cores might induce that some of the cores (e.g., the processors) send out requests, while others transmit responses (e.g., the memory controllers). In this example, the protocol determines that two types of messages exists. It provides the destinations of these messages. It also determines that, e.g., a response is injected only when a request has arrived. In this case, the application-layer protocol causes a *message dependency* between requests and responses [68].

Network Layer

The second layer considered in this thesis is the network layer. In the OSI model, this layer coincides with both the network and the transport layer. Each core is connected to a *processing node* (see Figure 1.3). This node is able to inject messages received from the core into the network, and to remove messages sent to the core from the network. Any message arriving at the processing node is sent through a switch.

Switches are connected to each other via *channels*. Each channel has a certain capacity to store messages. Channels are the only components in the network that buffer messages. Generally, channels are directed, i.e., they are used to transmit messages from switch to switch unidirectionally.

A switch applies arbitration to determine which of their in-going channels is served. A *routing function* decides where the message is sent to. It can choose between one of the channels going out of the processing node, or the local outgoing channel leading to the core. The set of possible channels to which a message can be routed in one step is called the set of *next hops* of the message. Each time a message moves from channel to channel, the processing node at the end of its current channel decides a set of next hops and selects one that is able to receive the message. Routing can be deterministic, in which case each message has at most one next hop. This implies that routes are static and can be precomputed. In contrast, adaptive routing may supply multiple next hops per message. In case of adaptive routing, a selection function determines which of the next hops is taken [44]. Note that in case of adaptive routing at each time the set of next hops is static, i.e., the routing function itself does not change. However, since it supplies multiple next hops of which only one is to be selected, the route that a message can take from source to destination can vary.

We assume there is some atomic unit of transfer called a flow control digit (*flit*) [36]. The size and the type of this unit depend on the type of network. Messages are transformed into flits before they traverse the network. A message may consist of one or more flits, but each flit belongs only to one message.

Link Layer

The lowest layer considered in this thesis is called the link layer. This layer contains the *transfer protocol* between two nodes. This protocol determines under which

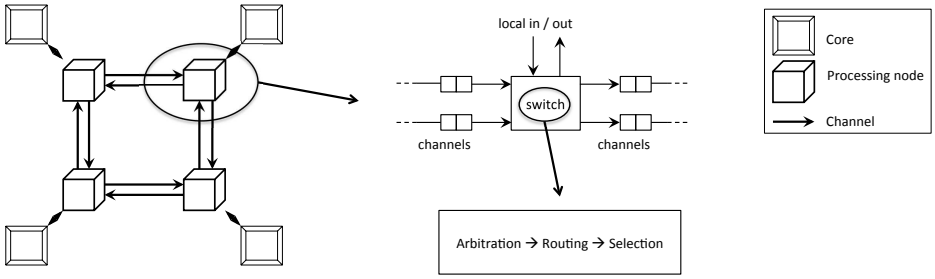


Figure 1.3: *Processing nodes in a network.*

conditions a message may move from channel to channel.

For example, a network may have wires between each pair of connected channels to facilitate a handshaking protocol. When the target channel is ready to receive, it will send out a signal on such a wire. The transmitting channel does the same when it is ready to send. The transfer protocol states that if both signals are high, a messages moves.

More complicated protocols can be applied. A network with credit-based flow control has extra wires to count the messages in the network. It prevents messages from moving to their next channel if a certain bound has been reached, even if the next channel has free buffers. Also, the transfer protocol may take care of synchronization between messages in the network. In this case, a message is moved only if some other message(s) in the network have reached a certain channel.

1.3 Network Layer Isolation versus Integration

The main structure of this thesis is based on two points of view on the network layer. In the first view, we formulate assumptions which abstract away from both the application layer and the link layer. This yields the *isolated network model*. In contrast, the *integrated network model* considers the three layers all at once. We first informally introduce both points of view.

In the isolated network model, the assumptions on the application layer state that cores are homogeneous and fair. As for the link layer, we assume that the transfer protocol moves a flit towards a next hop if this next hop is *available*, i.e., if the next hop is able to store the flit. This abstracts away among others counters or synchronizations. As a result, the isolated network model focuses on routing and topology. The isolated network model is based on the following assumptions:

- There is only one type of messages.
- Each core may send messages to all other cores.
- At each core, injection is fair, i.e., given some destination d each core will always eventually want to send a message to d.

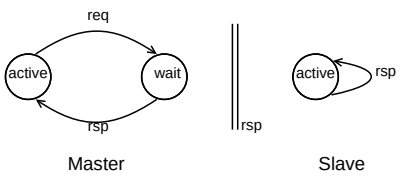
- At each core, consumption is fair, i.e., if a message arrives at a switch connected to the destination core of the message, it will eventually be consumed. This is commonly referred to as the consumption assumption.
- The size of messages is always finite.
- A flit moves, if there exists a next hop that has available capacity to store the flit.

While the isolated network model abstracts away from details, the integrated network model incorporates all details concerning all three layers. The integrated network model encompasses among others the behavior of the cores, different message types, the interfaces between the cores and the processing nodes, the network topology, the routing logic, the injection method, and the transfer protocol between channels.

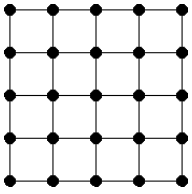
With two examples, we illustrate the different facets of both points of view.

1.3.1 Example 1

Consider the following simple application-layer protocol: a set of masters (e.g., processors) and slaves (e.g., memory controllers) communicate through request and response messages. A master sends a request to a slave and waits for the slave to return a response. Figure 1.4a shows the corresponding finite state machine. We use the handshaking operator $\|_\alpha$ from Baier and Katoen [5]. At the network layer, masters and slaves are organized in a two dimensional mesh (Figure 1.4b). Messages are routed using XY routing [103]. This routes messages first in the horizontal direction to the right column and then vertically towards the right row. As for the link layer, messages move whenever there is an available next hop.



(a) *FSM for protocol with one master and one slave*



(b) *Two dimensional mesh topology*

Figure 1.4

Taken in isolation, all components are deadlock-free. The protocol at the application layer is obviously deadlock-free. Masters wait for slaves, but slaves never wait and therefore there is no terminal state. The network is deadlock-free, as a 2D mesh with XY routing induces no circular dependencies [36]. There also are no circular message dependency, as a response can be generated by a node receiving a request, but not the other way around [68]. We show how deadlocks still may emerge from these deadlock-free components, depending on among others the layout of the masters and slaves.

Consider the layout in Figure 1.5 where every odd (even) column is filled with masters (slaves). Each request packet has a source s (the master where the response needs to be directed to) and a destination d (the slave where the request needs to be directed to). Each response package has a destination d only. A deadlock can occur when responses get blocked by requests and cannot arrive at their destination.

Figure 1.5 shows the smallest deadlock possible in this layout. Requests injected at Master 0 and destined for Slave 3 are heading east. At Slave 3, these requests are turned into responses destined for the original source, Master 0. These responses are blocked by western bound requests injected by Master 2. These requests are turned into responses heading east at Slave 1. A circular wait has been created, and no message has an escape possibility out of this cycle. The combination of the routing logic with message dependencies has caused a deadlock.

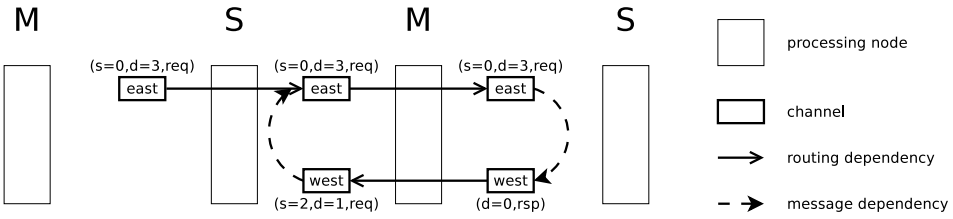


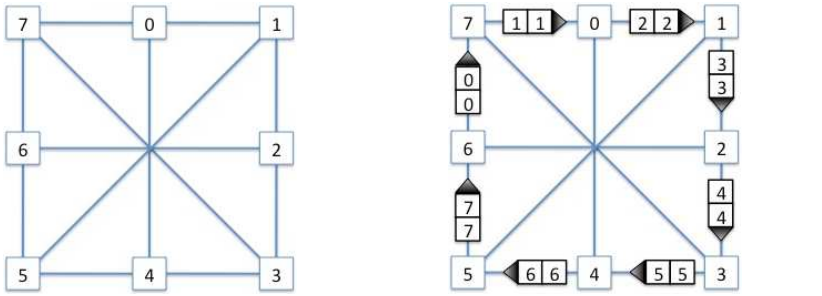
Figure 1.5: *Message-dependent deadlock in a 4x1 mesh. Nodes are identified by their x-coordinate.*

Now consider the layout where all masters (slaves) are put on the left (right) side of the mesh. In such a layout, requests can be blocked by responses and other requests. This introduces new dependencies with respect to the routing dependencies. However, these new message dependencies cannot introduce a cycle. This is implied by the fact that a western bound message never needs to wait for a vertical channel or an eastern channel. Any western bound message is always a response, as masters are on the left side of the mesh. The XY routing logic ensures that a western bound message is not routed north or south. The only way a western bound response can be blocked is by another western bound response. Eventually, some western bound response will arrive at its destination. As it is a response, it will be consumed and no further message dependencies will occur. Since a western bound message can wait only for western channels, no cycles occur. In this layout the network is deadlock-free.

1.3.2 Example 2

In this example, we abstract from the applications running on top of the network. The cores are simply assumed to be homogeneous. At the network layer, we consider the Spidergon NoC with shortest path routing [32] (see Figure 1.6a). Without further modification, the chip suffers from routing deadlocks. A deadlock can occur if each clockwise channel going out of processing node n is filled with messages destined for node $n + 2 \mod N$ with N the total number of processing

nodes (see Figure 1.6b).



(a) *Spidergon ring topology*

(b) *A deadlock. Each channel has capacity to store two messages.*

Figure 1.6: *Spidergon of STMElectronics*

However, whether this deadlock actually occurs can depend on the link layer. For example, a credit-based flow control can make the network deadlock-free. This transfer protocol moves messages towards a channel only if the total number of messages in the network is below a certain bound B. The described deadlock can occur only when there are at least kN messages in the network, with k the capacity of the channels. Consequently, the transfer protocol ensures deadlock freedom for any B less than kN.

1.3.3 Isolated versus Integrated

Example 1 shows that a network can be deadlock-free when considered in isolation, whereas considering the system integrally yields deadlocks. Conversely, Example 2 shows that a network that is considered to have deadlocks when viewed in isolation, can actually be deadlock-free when the system in analyzed in its entirety.

Considering the network layer in isolation has been common practice for many years. More so, it is the entire purpose of the OSI model to facilitate a divide-and-conquer approach. First of all, dealing with the huge complexity of the whole system is practically infeasible. Secondly, separating the network from the applications running on top of it allows a modular approach. This separation is often even necessary, when during the design of a communication network it is not known which specific applications run on top of it. In such cases, correctness of the network can only be assessed under generic assumptions on the behavior of the possible applications. This approach has the additional advantage that design and verification efforts are reusable: they are correct for any application that satisfies the assumptions. Thirdly, even though it is the case that when analyzing a network in isolation yields a deadlock this deadlock might not actually occur, the knowledge that the network has deadlocks when considered in isolation is still valuable. This knowledge tells a designer that additional precautions are *required* to ensure deadlock freedom, i.e., that it is necessary to add deadlock-preventing elements to either the application or the link layer.

The greatest disadvantage of the isolated network model is that it is a very abstract model of communication networks. At the application layer it does not

allow different types of cores, or different types of messages. At the link layer it does not allow counters, scoreboards, or synchronizations. Therefore, the result of the analysis does not provide trustworthy results: deadlocks are not necessarily actual deadlocks, and deadlock freedom does not ensure actual deadlock freedom.

The integrated approach has the obvious advantage that the result is more reliable: if the system is proven deadlock-free while considering all three layers monolithically, deadlock freedom of the entire system can be stated with a high degree of confidence. Also, the integrated network model is not restricted in any way: it allows for all kinds of applications such as cache coherency and master/slave protocols and for all kinds of flow control such as synchronizations, duplications, and packet transformations.

The major disadvantage of the integrated network layer model is the enormous increase in complexity. Also, results of a verification effort are not reusable since the effort has been done taking all facets of the system into account.

1.4 Contribution of this Thesis

This thesis consists of four parts. Part I, the preamble relates the contributions of this thesis to the current state-of-the-art and provides background information required for the remaining three parts. The contents of the remaining parts are based on fourteen publications, including five journal articles, six peer-reviewed conference papers, and three peer-reviewed workshop papers.

In Part II we apply the ACL2 theorem prover to prove correctness of on-chip interconnects. This part contains two major contributions.

1. We formalize a novel notion of correctness for NoCs.
2. We extend GeNoC, a formal theory of communication networks in the ACL2 theorem prover, with this new notion of correctness. We formulate a set of assumptions and prove a generic theorem which states that *any* interconnection network that satisfies the assumptions is correct. GeNoC will be discussed in detail in the next chapter.

Part II is based on the following publications:

- Freek Verbeek and Julien Schmaltz. *Easy Formal Specification and Validation of Unbounded Networks-on-Chips Architectures*. ACM Transactions on Design Automation of Electronic Systems (TODAES), volume 17 (issue 1), pages 1:1–1:28, January 2012.
- Freek Verbeek and Julien Schmaltz. *Formal Specification of Networks-on-Chips: Deadlock and Evacuation*. Proceedings of Design, Automation and Test in Europe (DATE'10), pages 1701–1706, March 2010.

One of the conclusions presented in Part II is that the absence of deadlocks is the most difficult assumption required to prove correctness. The remainder of this thesis is concerned with establishing deadlock freedom of communication networks.

Part III considers deadlock verification in the isolated network model. It presents the following contributions:

3. We formalize necessary and sufficient conditions for deadlock freedom of communication networks. A careful analysis of existing conditions is presented to show the relevance of our new conditions.

4. We present the tool DCI2 (for: Deadlock Checker In Designs of Communication Interconnects), which automatically decides whether these conditions hold for a given network. Additionally, it detects livelocks and various routing related issues under the *isolated* network model.

5. We prove deadlock detection NP-complete for wormhole networks.

6. Extensive experimental results are provided for non-trivial examples like adaptive fault-tolerant routing and for a chip with wireless transmissions.

The experimental results have been obtained in cooperation with Abdulaziz Alhussien and Nader Bagherzadeh from the University of California, Irvine (UCI). Part III is based on the following publications:

- Abdulaziz Alhussien, Freek Verbeek, Bernard van Gastel, Nader Bagherzadeh and Julien Schmaltz. *A Formally Verified Deadlock-Free Routing Function in a Fault-Tolerant NoC Architecture*. Proceedings of the 25th Symposium on Integrated Circuits and Systems Design (SBCCI'12), august 2012.

- Freek Verbeek and Julien Schmaltz. *Automatic verification for deadlock in networks-on-chips with adaptive routing and wormhole switching*. Proceedings of Networks-on-Chips Symposium (NOCS'11), pages 25–32, May 2011.

- Freek Verbeek and Julien Schmaltz. *A Fast and Verified Algorithm for Proving Store-and-Forward Networks Deadlock-Free*. Proceedings of The 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP'11), pages 3–10, February 2011.

- Freek Verbeek and Julien Schmaltz. *On Necessary and Sufficient Conditions for Deadlock-Free Routing in Wormhole Networks*. IEEE Transactions on Parallel and Distributed Systems (TPDS), volume 22 (issue 12), pages 2022–2032, December 2011.

- Freek Verbeek and Julien Schmaltz. *A Comment on "A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks"*. IEEE Transactions on Parallel and Distributed Systems (TPDS), volume 22 (issue 10), pages 1775–1776, October 2011.

Part IV presents a deadlock detection algorithm in the integrated network model. It contains the following contributions:

7. We provide a syntax to enable the description of micro architectural models of communication fabrics.

8. We provide an algorithm detecting deadlocks in communication fabrics described in this syntax.

Our algorithm is able to deal with many different facets of monolithical network verification such as message dependencies and different types of link layers. In essence, the algorithm reduces the decision of deadlock freedom to solving many SMT instances. The examples presented in this chapter can be proven deadlock-free with this algorithm. Part IV is based on the following publications:

- Freek Verbeek and Julien Schmaltz. *Automatic Generation of Deadlock Detection Algorithms for a Family of Microarchitecture Description Languages of Communication Fabrics.* Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'12). To appear.
- Freek Verbeek and Julien Schmaltz. *Towards the Formal Verification of Cache Coherency at the Architectural Level.* ACM Transactions on Design Automation of Electronic Systems (TODAES), volume 17 (issue 3), pages 20:1–20:16, June 2012.
- Freek Verbeek and Julien Schmaltz. *Hunting deadlocks efficiently in microarchitectural models of communication fabrics.* Proceedings of Formal Methods in Computer Aided Design (FMCAD'11), pages 223–231, November 2011.

1.5 The Role of ACL2 in this Thesis

The ACL2 theorem prover (see Section 2.3.1 for a short introduction and references for further reading) has played a vital role in the realization of this thesis. It has been used extensively to establish the validity of the definitions and proofs of many theorems. The focus of this thesis is on the theorems and the algorithms, but *not* on their verification in ACL2. Any reader interested in details on the ACL2 proofs can find these in the following publications:

- Freek Verbeek and Julien Schmaltz. *Proof Pearl: A formal proof of Dally & Seitz' necessary and sufficient condition for deadlock-free routing in interconnection networks.* Journal of Automated Reasoning (JAR), volume 48 (issue 4), pages 419–439, April 2012.
- Freek Verbeek and Julien Schmaltz. *Formal verification of a deadlock detection algorithm.* Proceedings of the International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'11), pages 103–112, November 2011.
- Freek Verbeek and Julien Schmaltz. *Proof Pearl: A formal proof of Duato's condition for deadlock-free adaptive networks.* Proceedings of Interactive Theorem Proving (ITP'10), pages 67–82, July 2010.
- Freek Verbeek and Julien Schmaltz. *Formal Validation of Deadlock Prevention in Networks-On-Chips.* Proceedings of the International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'09), pages 128–138, May 2009.

Figure 1.7 provides an overview of the ACL2 proof effort. The arrows do not represent implications, but stand for the structure in which the theorems have

been proven. The upper theorems and the definitions used to prove them have been used in proofs of lower theorems. Theorems 3.1, 3.2 and Corollary 3.1 are part of our extension of the GeNoC framework. The proofs in Sections 4.2 and 4.3 are examples of proofs established applying GeNoC. Theorems 6.1, 6.2, and 7.2 and Lemma's 6.4, 6.6 and 6.9 formulate correctness of our necessary and sufficient conditions. The proofs of these theorems rely upon the GeNoC framework. Finally, Theorems 7.1 and 7.3 formulate correctness of our algorithms, i.e., that they correctly check whether the necessary and sufficient conditions hold for some particular network.

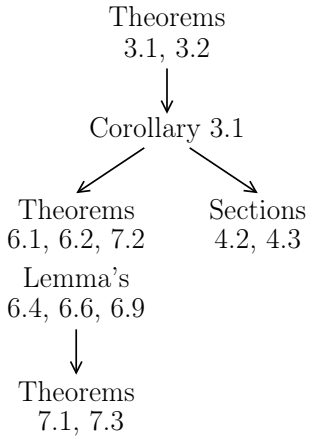


Figure 1.7: *Mechanically proven theorems*

All the theorems in Part IV have *not* yet been verified mechanically. Additionally, our proof that deciding deadlock freedom of wormhole networks is co-NP-complete, Theorem 3.3 and Lemma 6.5 have not been formalized in ACL2.

The total proof effort related to this thesis consists of 39.321 lines of ACL2 code. This includes 3446 theorems (ACL2 defthm statements) and 858 definitions (ACL2 defun statements). ACL2 performs – while proving all the theorems – about 2.352.273.987 prover steps. All books are available upon request.

The added value of verifying all these theorems in ACL2 is twofold. First, it helped with getting all definitions completely correct. As example, the definition of a valid configuration in a wormhole network contains many subtleties. Section 6.7.2 (see Page 94) will show that the definitions as they were used in existing literature were incorrect. It was due to our formalization in ACL2 that we were able to formulate accurate and precise definitions. Secondly, the ACL2 theorem prover has greatly helped while *designing* the algorithms presented in this thesis. Our initial proof effort started with flawed algorithms. The final versions presented in Chapter 7 are the result of a mutual recursion between reformulating the algorithms and proving theorems about them. The algorithms contain a post-processing step, which is only required when for some very specific network, some specific *trace* of the algorithms yields an erroneous result. We would not have found the necessity of this post-processing step without proving correctness of the algorithms in ACL2.

Chapter 2
Advances to the State-of-the-art

This chapter discusses background information to facilitate the reading of the remaining parts of this thesis. We present the current state-of-the-art in formal verification of interconnection networks, with a focus on the architectures used on-chip. Whenever possible, we clearly show how this thesis advances the current state-of-the-art. After providing background information on interconnection networks, we give an overview on research related to deadlocks in communication networks. Finally, we provide an overview of the mechanical verification efforts related to communication networks.

2.1 Communication Networks

During the '00s, a multi-core shift occurred [94, 58]. In 2000, it was already possible to design SoCs with dozens of mutually communicating IP blocks. Following Moore's Law, more and more transistors could be put on a chip each year [98]. In 2009, the Intel single-chip cloud was able to run 48 general purpose – fully programmable – processing cores on one single die [74].

This revolution evoked a discussion on the interconnect between the cores. In 2000, Guerrier and Greiner discuss that the increase in complexity caused by multiple systems on a chip requires a shift from the traditional bus towards an on-chip network [67]. In the same year, Hemani et al. present an NoC, where resources communicate using addressed packets routed to their destination via a communication fabric [72]. Dally and Towles present an example of an on-chip network and discuss several challenges in their design [37]. In 2002, Benini and Micheli state that with more and more cores on a chip, the bus becomes the bottleneck in the overall performance of the system and coin NoC a new paradigm in chip design [8].

Interconnection networks, however, are not new. The importance of networking to communicate between processes was already recognized in the 70's. The ARPA network (the precursor of the Internet) triggered a great amount of research related to the performance, reliability and correctness of interconnects between computers [25][1]. During the 80's and 90's, the interconnection networks *inside*

[1] This paper describes a deadlock on the "Internet" between UCLA and Stanford University.

multicomputers were widely studied [36, 44]. NoCs brought interconnection networks to the level of integrated circuits. What first took hundreds of square meters (e.g., the ASCI Red Super Computer reaching 1 teraflops in 1996), is now embedded on a single die (i.e., Intel's Teraflops Research Chip with 80 cores).

Wolf recognizes two crucial differences between the early computer networks and the modern on-chip ones [145]. First, embedded systems require predictable performance. Due to the close cooperation between cores, bounds on the latency and throughput become crucial for the performance and correctness of the system. Secondly, constraints posed by power and energy become an issue on every level of abstraction. For example, whereas the interconnect between computers can simply drop and resend messages in order to resolve a deadlock, this becomes very costly in an on-chip network. In addition, Vermeulen et al. note that the topology of an on-chip network is static, whereas off-chip networks often dynamically add and remove links [143].

Switching in NoCs

Generally, three types of switching are being used: circuit, packet[2] and wormhole switching. Packet and wormhole switching are the most commonly used [112, 107, 140]. Therefore, this thesis is not concerned with circuit switching.

Packet Switching

In a network with packet switching, messages are packetized before being injected. A *packet* consists of a payload and a header. The payload is the data of the message that is to be transmitted. The header contains the destination of the packet. A packet is the atomic unit of transfer.

As a packet contains all the information needed to route the packet through a network, it can move autonomously from channel to channel. At each router in the network, one or more next hops are determined. If one of these next hops is able to accept the packet, the packet is forwarded to that next hop. There is no reservation or prebooking of channels.

A crucial feature of packet networks is that the size of the packets is fixed and that the network is set up accordingly. A channel with capacity n can store exactly n packets. A packet is stored completely in a buffer of a channel. If it is routed towards a next channel, it is removed from its old position and moved integrally to a buffer of the next channel.

Since packets are stored and require new channels to be able to store them before they can move, contention may occur. A channel is available if it has at least one empty buffer. A packet is blocked if all its next hops are unavailable.

Channels can be implemented in two ways: *queues* and *central buffers*. With queues, the order in which packets arrive at the channel is relevant. Only the packet that arrived first, i.e., the packet at the head of the queue, is considered for forwarding to a next hop. With central buffers, any packet in the channel can be chosen to be forwarded to a next hop.

[2] In this thesis, packet switching denotes store-and-forward switching.

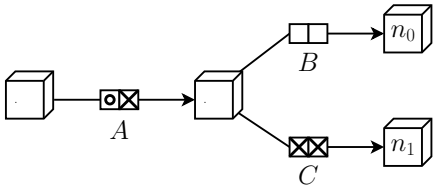


Figure 2.1: *Network with packet switching. The circle (cross) packets are destined for n_0 (n_1).*

To stress this difference, consider Figure 2.1. The network contains three channels all with capacity 2. Channel A stores two packets, one destined for processing node n_0 and one destined for processing node n_1. Channel C is full. Under queue semantics, none of the packets in channel A can move if the packet destined for n_1 arrived first. Only the packet at the head of the queue is considered, and this packet cannot move because channel C has no empty buffer. Under central-buffer semantics, the packet in channel A destined for n_0 can move.

Packet networks generally tend to have low contention, as packets do not occupy many resources. Adaptive routing can decrease contention even further, as packets move autonomously through the network. However, the storing of the packets in buffers makes latency unpredictable as it depends on the load of the network. Also, the size of the packets bounds the size of the messages that can be transmitted. If one message is divided into multiple packets, then some ordering mechanism has to ensure that packets are delivered in-order. For an in-depth discussion on packet switching, we refer to standard textbooks [38, 48].

Wormhole Switching

In a network with wormhole switching, messages are split into *worms* before being injected. A worm consists of an arbitrary but finite number of flits. Two types of flits exist. A *header flit* is a flit containing the destination of the message. A *tail flit*, is a flit containing a part of the message that is to be sent. Each message has one header flit, but typically many tail flits. It is common practice to further distinguish between data flits, end-of-worm flits or flits storing the length of the worm. In this thesis, however, this distinction is not necessary and all these flits are denoted simply as tail flits.

When injecting a message, the header flit is injected first and leads the way. The header flit is routed autonomously from channel to channel. Similar to a packet, at each router a set of next hops is determined and the header flit is sent to a next hop that is ready to receive it. As the tail flits do not contain any information on their destination, they follow the header flit in a pipelined fashion.

At each successive channel, flits are stored before they are forwarded to their next hops. Each channel with capacity n is able to store exactly n flits. Contention occurs when the header flit has to wait for its next hops to become available.

A channel is available to accept tail flits if it has free buffers to store these flits. A channel is available to accept a header flit only if *all* its buffers are empty. This is of importance for the pipelining process. If channels store flits belonging to different messages, there is no way to distinguish which flits belong to which

message. A channel with tail flits cannot even accept the header flit of its own message.

Consequently, a header flit is blocked if all its next hops are non-empty. A tail flit is blocked if the next channel in the worm is full.

The most important advantage of wormhole networks is that they allow for relatively small buffers, as messages can be split up into many flits. However, as messages may hold many resources simultaneously, contention can be higher. For more details, we refer to standard text books [38, 48].

2.2 Deadlocks in Communication Networks

A deadlock is a situation where a set of processes is permanently blocked and no progress is ever possible. This can occur due to a competition for finite resources or reciprocal communications. Classically a deadlock is associated with a circular wait between processes: each process holds a resource needed by the next process [134]. In the context of interconnection networks, processes are messages and resources are channels. A deadlock can occur as messages compete for available channels.

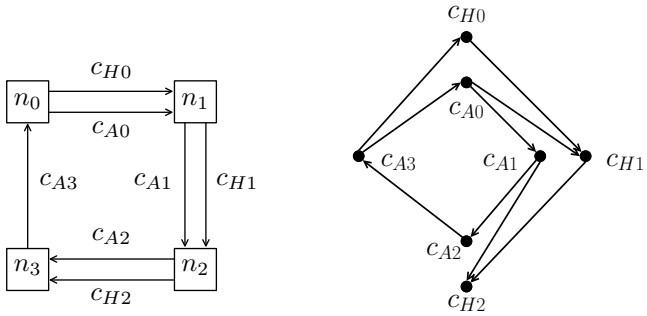
There are three ways to deal with deadlocks: avoidance, prevention, and detection. In networks where deadlocks are avoided, deadlock freedom is ensured dynamically. The on-chip switching has some look-ahead mechanism and bases its decisions on extra information. In contrast, deadlock prevention statically ensures deadlock freedom. The routes messages can take are restricted, e.g., in such a way that messages are not sent into a circular wait. In networks with deadlock detection, routing can be less restricted. An online deadlock detection and recovery mechanism deals with deadlocks that occur as a result of these relaxed routing schemes.

In typical on-chip networks, routing decisions must be taken in a few nanoseconds. On-chip deadlock avoidance is too costly and is generally not considered a solution for NoCs. Some on-chip deadlock detection mechanisms have been proposed by Pinkston [116] and Martinez-Rubio et al. [93]. The most practical way of dealing with deadlocks is to prevent them by designing deadlock-free routing functions. This has been a fruitful research area for many years [11, 35, 18, 61, 28, 132].

2.2.1 Necessary and Sufficient Conditions

This research has lead to a search for *generic* conditions ensuring that a routing function is deadlock-free [36, 43, 44, 128, 54, 138]. Typically, the dependencies between channels are captured by a *dependency graph*. Early work by Dally and Seitz has shown that an acyclic dependency graph is a necessary and sufficient condition for deadlock-free routing [36]. This original condition only applies to deterministic routing functions.

An interconnection network can have an adaptive routing function. If a message is blocked on its way, an adaptive routing function proposes an alternative next hop allowing further progress. To the best of our knowledge, Chen was the first to notice that a cyclic dependency is not sufficient for deadlock in 1974 [25]. Cypher and Gravano prove a routing function deadlock-free that allows circular dependencies in a packet network [34]. Duato was the first to propose necessary



(a) *Interconnection network* **(b)** *Channel dependency graph*

Figure 2.2: *Example of cyclic dependencies without deadlock[48]*

and sufficient conditions for deadlock-free routing in adaptive networks [44, 45]. He noticed that alternative paths could be used to *escape* deadlock situations and that a cyclic dependency is not a sufficient condition to create a deadlock [43]. He used Example 2.1 to demonstrate this [48].

Example 2.1 Consider the interconnection network in Figure 2.2a. Routing is defined as follows: when routing a message from source n_i to destination n_j, the routing function always returns channel c_{Ai}. It returns channel c_{Hi} only if $j > i$. The c_{Hi} channels do not form a dependency cycle, implying that they will always eventually become available. The c_{Ai} channels do form a dependency cycle. However, even if all channels of this cycle are unavailable, messages in node n_0 can always *escape* the cycle by using channel c_{H0}. After this, the messages in the cycle can progress.

Based on his intuition, he defined and proved a condition capturing the fact that an adaptive network can still be deadlock-free even in the presence of cyclic dependencies between channels [44]. This was a breakthrough in the field as it enables a dramatic reduction in the number of resources to implement fully adaptive routing networks. It is also counterintuitive as it seems that a circular wait is not sufficient for deadlock. Duato's work was not easily accepted by his peers. On Duato's webpage one can read [46]:

> Only a complex mathematical proof can show that deadlock freedom can be guaranteed if certain conditions are met. This research was so disruptive when it was developed that it was rejected by several peers and considered to be incorrect, even by the most prominent researchers at that time. However, it was finally accepted and several well-known researchers developed their own version of this theory.

Several more papers have been devoted to generic necessary and sufficient conditions for deadlock-free routing. Duato's condition holds for routing functions which are based on the current location of the message and its destination. Schwiebert and Jayasimha present a condition that holds for routing functions which also base their decisions on the direction from which the message came [128]. Fleury and Fraigniaud extended Duato's result to a broad class of routing functions [54]. Tak-

tak et al. were the first to present a condition that can be checked automatically in polynomial time [138]. Their condition is logically equivalent to Duato's one. These conditions will be discussed in more detail in Chapter 6.

All these papers are devoted to wormhole switching. Wormhole networks are generally more prone to deadlock than packet networks. Messages can hold multiple channels at once. Channels are unavailable more often, as they can only store flits belonging to one message. Both types of network require different definitions and conditions for deadlock freedom. Duato also defined his condition for packet networks [45]. To the best of our knowledge, this is the only necessary and sufficient condition for adaptive deadlock-free routing in packet networks.

The contribution of this thesis: Chapter 6 will present new necessary and sufficient conditions for packet and wormhole networks. The proofs of correctness of existing conditions are often complex, highly abstract and counterintuitive. Therefore, we have applied the ACL2 theorem prover to mechanically prove correctness of our conditions. We will show that Duato's seminal condition for wormhole networks is not necessary and sufficient. Our condition is the first static necessary and sufficient condition for deadlock freedom in wormhole networks.

2.2.2 Determining Deadlock Freedom

Once necessary and sufficient conditions for the absence of deadlocks have been established, a natural next question is how to determine whether some network actually satisfies such a condition. In other words, how can we use these conditions to establish deadlock freedom of a given design? The condition of Dally and Seitz can easily be by checked by computing the dependency graph and performing a linear search for cycles [33]. For adaptive routing however, this is more complicated as cycles are not necessarily deadlocks.

The most straightforward approach of determining that some network satisfies a condition is to just prove this by hand. Such a proof can be done mechanically or with pencil and paper. Advantage is that such proofs can be done leaving the size of the network parametric. However, such proofs become tedious very quickly. They are very time-consuming and require a lot of user interaction.

Another approach of determining that the network is deadlock-free is to limit the design process using some *design methodology*. Duato, Silla, et al. present a design methodology based on Duato's condition [44, 132]. Consider a network and a routing function that are already known to be deadlock-free. New channels and routing capabilities can be added to this network as long as once a message arrives in an original channel, it cannot be routed towards new channels. Duato's condition holds for any network obtained in this way.

Various design methodologies are based on a *turn model* [61, 29]. This model restricts the routing function in such a way that no cyclic dependencies occur. It was initially defined for 2D-mesh topologies. Starobinksi et al. use the network calculus to generalize this methodology to general topologies [135]. Palesi et al. incorporate message dependencies, induced by a model of the applications running on the cores, into the dependency graph and provide a design methodology breaking all cycles [110].

These methodologies target static and fault-free networks. Duato, Lysne, et al. present a theory and a corresponding methodology to dynamically reconfigure a routing function in such a way that deadlock freedom is ensured [47, 90]. Their methodology applies to broad class of interconnection networks, both on-chip and off-chip.

An inherent disadvantage of such design methodologies is that they limit the designer. All these methods cannot be applied to arbitrary topologies and routing functions. Also, these methodologies are restrictive, in the sense that many deadlock-free designs are excluded. Most of these methodologies break all dependency cycles. As for adaptive routing a dependency cycle is not necessarily a deadlock, this is very limiting. Finally, these methodologies are manual and therefore error-prone.

A third approach of determining deadlock freedom is to add new channels – either virtual or physical – to a given design [36]. Seiculescu et al. automatically detect cyclic dependencies and determine where new channels can be added to break these [129]. Their approach is based on the sufficient condition that an acyclic dependency graph ensures deadlock freedom.

In 2008, Taktak et al. presented a different approach. They presented a dedicated algorithm to either detect deadlocks or prove deadlock freedom [137, 138]. The work of Taktak et al. will be discussed in more detail in Section 7.8 (see Page 129).

The contribution of this thesis: Our approach is similar to that of Taktak et al. Chapter 7 will present two fully automatic decision procedures for deadlock freedom in both packet and wormhole networks. Our wormhole algorithm checks a modified version of our necessary and sufficient condition. This version is only sufficient and therefore our algorithm may identify false deadlocks. We will show that deciding deadlock freedom of wormhole networks is co-NP-complete. This implies that the polynomial algorithm of Taktak et al. does not check a necessary and sufficient condition as well. Indeed, as the correctness of the algorithm of Taktak et al. is – indirectly – based on Duato's condition, it is subject to false negatives. In contrast, the correctness of our algorithms has been established mechanically, using the ACL2 theorem prover. Our wormhole algorithm is one degree faster than that of Taktak et al. We will show that this improvement has a significant practical impact. As our tool is able to analyze millions of different network topologies efficiently, we are able to apply it to the verification of a fault-tolerant routing function in a mesh of 400 cores. This degree of scalability is not reachable with state-of-the-art tools.

2.3 Mechanical Verification of Interconnects

Several specific NoC architectures have been studied using model checking [122, 124, 27], theorem proving [60] or combinations thereof [1]. Regarding formal proofs of deadlock prevention, Gebremichael et al. formally prove a sufficient condition for the Æthereal protocol of Philips in a packet-switched network [60]. The main property that has been verified is the absence of deadlock for an arbitrary number of masters and slaves. These works target very specific designs described at a low

level of abstraction. Goossens recognizes the need for more parametric verification: "Ideally, deadlock freedom would be proven for any instance of the NoC." [63]

A different methodology is presented by Chatterjee et al. of the Intel Corporation. Recently, Chatterjee et al. proposed xMAS as a formal description language for microarchitectures [24, 23]. This language is based on a restricted set of primitives with well-defined semantics. The methodology consists of automatically generating inductive invariants and using this these to model check a design [22]. This way they can establish, among others, deadlock freedom. Chapter 9 will discuss the xMAS language in-depth.

This thesis applies theorem proving to do parametric proofs over NoCs. Since the mid 70's, interactive theorem provers have been designed to mechanically check formal and detailed proofs. Their development and application in various domains are active research fields. These proof assistants are used in projects about formalizing mathematics (e.g., the FlySpeck project [104, 106]) or in the verification of hardware and software designs (e.g., microprocessors [14, 55, 75], floating point units [123, 76, 10, 69], operating systems [87], entire computing systems [13]). The most popular tools are ACL2 [85], Coq [12], HOL [64], HOL-Light [70], Isabelle [105], and PVS [109].

To the best of our knowledge, the only work that targets parametric proofs over interconnection networks is by Schmaltz et al. [127, 21]. Schmaltz created a generic framework called *GeNoC* for reasoning about NoCs in the ACL2 theorem prover. We shortly introduce the ACL2 theorem prover and the GeNoC framework.

2.3.1 ACL2

ACL2 (for *A Computational Logic for Applicative Common Lisp*) denotes a language, a logic, and an interactive theorem proving system. It is the last of the Boyer-Moore family of provers and has been developed at the University of Texas at Austin. Currently, it is maintained by Kaufmann and Moore [85]. ACL2 has been used in academia to verify, for example, the mu calculus [86], theorems on real and complex numbers [59], and graph theory [86]. It has also been adopted by industry for the verification of, e.g., the floating point division microcode on the AMD-K5 processor [80], the X86-compatible microprocessor at Centaur [77], and verifying compilers at Rockwell Collins [115]. ACL2 has won the ACM System Software Awards in 2005.

The language of ACL2 consists of a superset of a subset of common LISP. Its logic consists of a classical first-order logic with induction. Axioms are added which assign interpretation to common symbols from the language such as `car` (i.e., the first element of a list) and `natp` (i.e., a recognizer for natural numbers). The result is called the *ground zero theory*. This theory allows ACL2 to reason over common LISP symbols.

Extension principles allow ACL2 to extend the ground zero theory with additional axioms, and are key to the use of ACL2 to reason about any non-trivial theorem [84]. There are three different extension principles. The *definitional principle* can extend the theory with axioms that assign a definition to a new function. As long as this function is terminating and does not contain fresh variables, sound-

ness of the theory is preserved. The *defchoose principle* allows the introduction of Skolemized formulas, which brings the abilities of quantifiers to the ACL2 logic. Thirdly, the *encapsulation principle* extends the theory with functions without a definition. Since this thesis does not present details on ACL2 proofs, we will not further introduce the first two principles. The encapsulation principle, however, is key to the GeNoC methodology. We explain it in more detail.

Encapsulation allows the introduction of a fresh function symbol f that satisfies a certain set of theorems $t_0, t_1, \ldots t_n$. As example, one might add the function symbol `sort` to the current theory, with as axioms `ordered(sort(x))` and `perm(sort(x),x)`. Here we assume that `ordered` and `perm` have already been defined with their obvious interpretations. Function symbol `sort` is called a *constrained function* and the theorems $t_0, t_1, \ldots t_n$ are called *proof obligations*.

Extending a theory with a constrained function can possibly make the theory inconsistent. Therefore, the following conditions have to be satisfied before a constrained function f with axioms $t_0, t_1, \ldots t_n$ can be admitted to the theory:

- Symbol name f must be fresh.
- It is logically consistent to add a witness function f_w to the current theory, and the theorems $t_0[f = f_w], t_1[f = f_w], \ldots, t_n[f = f_w]$.

In other words, we must provide a witness function f_w for which all the proof obligations are proven. This witness is local, meaning that it will not actually be admitted to the current theory. Its only purpose is to ensure the soundness of the extension. Constrained function f has no definition when it is admitted.

The crux of encapsulation is that any theorem that holds for a constrained function, also holds for any function that satisfies all the proof obligations. This is stated by the rule of inference called *functional instantiation*. Encapsulation and instantiation enable the following methodology:

1. First, a *generic theorem* is proven, e.g., the idempotence of `sort`:

   ```
   sort(sort(x)) = sort(x)
   ```

2. An *instantiation* is made of the generic function. As example, one might program merge sort, yielding a function `mergesort`.

3. For the instantiation, all proof obligations must be discharged. That is, we must establish:

   `ordered(mergesort(x))` and `perm(mergesort(x), x)`

4. By functional instantiation, we can now safely conclude that our generic theorem also holds for our instantiation. That is, without any further proving we have established:

   ```
   mergesort(mergesort(x)) = mergesort(x)
   ```

2.3.2 GeNoC

The GeNoC framework (for *Generic Network-on-Chip*) provides a methodology to do parametric proofs over NoCs. The proof methodology applies the concepts of encapsulation and instantiation to reason about the generic object "Communication Network". This generic network will be referred to with the bold capital **N**. Schmaltz et al. formulate proof obligations and use them to prove generic theorems. This yields a set of proof obligations and a theorem of the following form:

$$\mathrm{PO1}(\mathbf{N}), \mathrm{PO2}(\mathbf{N}), \ldots \quad \models \quad \mathrm{correctness}(\mathbf{N}) \tag{1}$$

This theorem allows a user to do proofs by instantiation. If we want to prove correctness of an actual communication network n, the proof obligations must be discharged.

$$\mathrm{PO1}(n),\ \mathrm{PO2}(n),\ \ldots \tag{2}$$

From this, it automatically follows that our network n is correct:

$$(1),\ (2) \quad \Longrightarrow \quad \mathrm{correctness}(n)$$

The value of the methodology enabled by GeNoC lies in the fact that even though proving a generic theorem can be difficult and require a lot of user interaction, this work is done once-and-for-all. As it is a generic theorem, it holds for all instances and it needs not be proven again. The easier it is to discharge the proof obligations, the greater the value of the methodology. For example, proving that a complicated sorting function s is idempotent might be difficult. The generic theorem presented in the previous subsection reduces this proof to a proof that s orders and permutes. Similarly, GeNoC reduces difficult proofs of correctness to the discharging of simple proof obligations.

The central part of the methodology is a generic theorem such as Equation 1. Such a theorem requires the study of four different problems.

1. The theorem mentions a generic communication network **N**. We need to know what a generic network is. This entails the question which constituents are common to all communication networks.

2. We need to define correctness.

3. It must be determined what proof obligations we may assume. If the assumptions we make are too strong, this will limit the applicability of the methodology. If they are too weak, we cannot prove anything significant.

4. The generic theorem must be proven, i.e., we need to show that the proof obligations imply correctness.

GeNoC deals with all of these problems. It contains the definition of generic communication network **N**, consisting of an injection method, a routing function, and switching policy. The GeNoC framework will be presented in detail in Part II.

GeNoC has been used to prove correctness theorems over a wide range of NoCs. Borrione et al. made an instantiation of the HERMES NoC [99] with XY routing [19, 20]. Schmaltz formalized double y-channel routing in a 2D mesh [125]. Schmaltz et al. used GeNoC to prove functional correctness of the Spidergon NoC architecture [32] with a ring topology and shortest path routing [126]. Helmy et al. have made an instantiation of the Nostrum chip with fully adaptive hot-potato routing [71]. Tsiligiannis and Pierre present an approach complementary to GeNoC that focuses on the verification of register transfer level implementations of communication interconnects [141].

In the current version of GeNoC [127, 21], the routing function is required to be deterministic and routes must be computed from the current position of the message all the way to its destination. Even if non-minimal and adaptive routing can be represented, the formalization is restricted to special cases and the verification is cumbersome. The correctness criterion currently supported by GeNoC is restricted to a safety property. The theorem states that messages reaching a destination will reach the expected destination without modification of their content. This is rather weak, as the theorem holds trivially if no message ever reaches a destination.

The contribution of this thesis: Chapter 3 extends the GeNoC framework with a new notion of correctness and a new theorem stating this correctness. This correctness property – which will be referred to as productivity – is a strong liveness property stating that always eventually messages are injected and always eventually messages correctly arrive at their destination. We argue that productivity is a relevant correctness property by showing that it is commonly used as an implicit assumption in related work. The generic network is further generalized to support adaptive routing, which no longer needs to be computed all the way to the destination. Chapter 4 provides a complete instantiation of a 2D mesh NoC with adaptive west-first routing, FIFO arbitration, wormhole switching and time-based injection.

Part II

Proving Productivity of Communication Networks

Chapter 3
GeNoC for Productivity Proofs

In this part, we extend the GeNoC framework for proving correctness of communication networks. Up to now, GeNoC supported the proof of partial functional correctness only. This chapter informally introduces a new correctness criterion. We call our correctness criterion *productivity* and relate it to other correctness criteria such as deadlock freedom and liveness. We break down a proof of productivity into a proof of several smaller proof obligations. The next chapter illustrates our approach on an example. We prove productivity of a realistic Network-on-Chip. For this particular example, 86% of the entire proof effort can be derived automatically using GeNoC. Chapter 5 reflects on both our approach and our correctness criterion.

3.1 Correctness of Communication Networks

Intuitively, a communication network is correct if any message can always eventually be injected into the network and if any injected message will always eventually be correctly consumed. A correct network behaves – from the point of view of the application layer – just like a point-to-point network with nondeterministic delay. This notion of correctness supports the design of distributed applications without considering the details of the underlying communication structure.

We capture this type of correctness in the property we named productivity. It states that:

1. any finite list of messages can eventually be injected;
2. for any injected message a bound can be defined on its time in the network, *independent* of the messages that are still to be injected;
3. any message is eventually consumed correctly, i.e., at its desired destination and without modification of its contents.

Our definition of productivity considers finite executions only. In Section 5.1 we show that this is indeed sufficient to capture correctness.

Deadlock freedom is often considered as correctness property of interconnects. Some consider deadlock freedom a global property, that is, a deadlock is a situation in which all messages are blocked [48]. Some consider deadlock freedom a

local property, stating that it is not possible for any message to be permanently blocked [73, 65]. Productivity encapsulates both types of deadlock freedom. In case of a deadlock some set of messages have been injected in the network, but they never arrive at their destination as they are permanently blocked. A deadlock can also cause input queues to be permanently blocked, preventing messages from being injected.

Productivity also subsumes livelock freedom and starvation freedom. Several correctness properties on communication networks are considered in this thesis. We present a short overview to show how the different properties relate to each other.

Deadlock freedom A network is deadlock-free if there is no reachable configuration in which some set of messages is permanently blocked. Note that as we consider deadlock prevention (see Section 2.2 on Page 20) a network is considered deadlock-free only if *no* execution leads to a deadlock. Deadlocks can be caused by incorrect routing, by injecting too many messages, or by contention caused by switching.

Livelock freedom A livelock is a scenario in which some message moves around infinitely in the network. A routing function can cause livelocks by sending messages into cycles.

Starvation freedom A starvation scenario occurs when a message wants to acquire some resource but is denied access infinitely often. Starvation is generally caused by unfair arbitration at merges.

Liveness of injection Injection of a network is live if always when the network is empty and there is a message to be injected, eventually a message will be injected. This property is trivial, but necessary for a formal proof of productivity.

Functional correctness Functional correctness is a safety property stating that *if* a message is moving through the network it moves in a correct way, i.e., it follows a correct path between connected resources, its contents are not modified and *if* it is consumed, it is consumed at its desired destination.

Evacuation A network is evacuatable if any injected message is always eventually consumed. Evacuatability does not state anything on functional correctness. For example, a network in which all messages are dropped after a fixed number of hops is evacuatable.

Local liveness A network is locally live if at all times no injected message can be permanently blocked. Any message will always eventually move from its current resource to a next one. Both absence of deadlock and absence of starvation are required for local liveness.

Reliability Reliability, or robustness, is a term commonly used in fault-tolerant related research [101, 133, 96, 119, 149]. Reliability is generally considered a probabilistic property representing the degree in which a system operates correctly in presence of faults. Reliability is not considered in this thesis,

but is mentioned here to emphasize the difference between reliability and productivity.

Productivity Productivity is the main property proven in this thesis and represents the intuitive correctness criterion that messages can always be injected and will alway correctly be consumed.

In-order delivery In-order delivery states that messages are consumed in some specified order. For example, in a network it may be required that requests are always consumed before responses.

Figure 3.1 provides an overview. The five upper properties are the primary correctness properties. They are all required to prove productivity. In-order delivery is a stronger notion of correctness than productivity. Reliability is a separate property independent of the others.

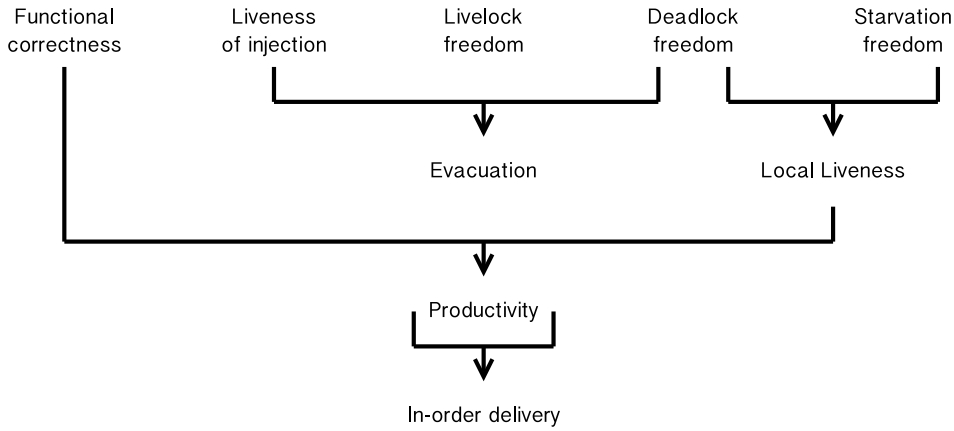


Figure 3.1: *Relation between correctness properties.*

3.2 Generic Communication Network

We prove a generic theorem that states that productivity is implied by several proof obligations. This proof is structured in three parts for functional correctness, evacuation and local liveness. Each of these properties has a section where we formalize the property, provide proof obligations, and prove a generic theorem stating that the proof obligations are sufficient. All these proofs revolve around the generic communication network, which will be referred to with **N**. This section introduces our generic model of communication networks.

3.2.1 Informal Overview

In a communication network, a set of *resources* can store messages. Messages are injected into resources, subsequently moved from resource to resource, until they can be consumed.

The generic network consists of constituents that can be found in all communication networks. Such constituents are called *generic constituents*. From these constituents, we build function $\mathbf{N_{beh}}$ that determines the behavior of the generic network $\mathbf{N}$. We prove, e.g., that this function never drops a message, that it never moves messages between unconnected resources and, ultimately, that it is productive.

We define the correctness properties using two sources of information. First, the current *configuration* stores all the information in the network, i.e., which messages are currently injected and which resources are held by these messages.

Besides storing the current state of the network, a configuration also stores *meta data*. This is data that is not actually present in the network, but which is required to formulate properties and prove theorems. For example, with each message some unique identifier is associated. In the configuration, each bit of information that is stored in some place in the network is linked to a message via this identifier. Other meta data includes which messages have arrived in previous steps and which messages are still to be injected.

Function $\mathbf{N_{beh}}$ takes as input the current configuration and the current meta data. It executes one step and recursively calls itself. It terminates when no progression is possible. This means that it terminates when either the network is empty and there are no new messages to be injected, or a deadlock has been reached.

We establish functional correctness by proving invariants over function $\mathbf{N_{beh}}$. For example, an invariant states that at any time, all messages have at least one way to reach their destination. Evacuation is formulated and proven using meta data. We prove that the list of consumed messages eventually equals the initial list of uninjected messages. Local liveness is proven by computing an upper bound on the time a message can wait in one resource.

Many of the proofs involve *measure functions* (also called ranking functions in the literature). A measure function is a function that returns a value that decreases – under some conditions – with each step. For example, we will define a measure function for the time it takes for a message to be injected, under the condition that the network is empty. The value of this measure function decreases, under some well-founded relation, as long as the network is empty. From several measure functions, we construct a measure proving termination of function $\mathbf{N_{beh}}$.

We let function $\mathbf{N_{beh}}$ record a *log* for each message. The log stores the progression of a message as it traverses the network. We define functional correctness using these logs. For example, we prove that it is an invariant that the logs of enroute messages always constitute valid connected paths in the network. We prove that uninjected messages have empty logs, and that the logs of consumed messages are not altered. From these theorems, it follows that messages always follow a correct route through the network.

Proof obligations which do not concern the behavior of the network, but meta data only, are not mentioned in this thesis. For example, we require proof obligations that ensure the logs are kept correctly. Also, some proof obligations that are considered trivial have been omitted. For example, a proof obligation states that at all times the current configuration is a syntactically valid data structure. These omitted proof obligations are used as assumptions in the proofs in this thesis. We

denote the set of implicit proof obligations with POι. All other proof obligations have been included in this thesis.

Generic functions, such as $\mathbf{N_{beh}}$, will be denoted by bold capital letters. Functions using meta data, such as the measures, will be denoted by bold Greek letters. The correctness properties that are proven are denoted in sans serif font, e.g., productivity is denoted by PROD.

The next section presents the generic network. The generic constituents are presented, together with function $\mathbf{N_{beh}}$. This section introduces notation used in the remainder of this part of the thesis. At the end of each section, a table will provide an overview of the notation, generic functions, and proof obligations presented in that section. The three next sections prove generic theorems related to functional correctness, evacuation and local liveness. We then formally define productivity and formulate our final generic theorem.

3.2.2 Formal Network Model

We formally introduce the concepts of resources, messages and configurations.

Resources

Network $\mathbf{N}$ consists of a set of resources R. Generic function **RGen** generates the resources given some instantiation specific parameters (e.g., the dimension of a 2D mesh topology). These parameters are implicitly provided to most of the generic functions mentioned in this paper. Each resource r has a certain *capacity*, i.e., a certain number of places to store data of messages, denoted $|r|$. We assume resources to be static, in the sense that neither the resources nor their capacities change during execution of the network.

Messages

Messages are the unit of communication at the network layer. When a message is injected into the network, it is wrapped into a data structure called a *travel*. A travel t is a formal representation of an injected message. A travel stores the progress of sending a message across a network. It is a tuple $\langle id, org, msg, l, d, log\rangle$ where id is a unique identifier, org is the resource where the travel is to be injected, msg represents the actual contents of the message, l represents the current location of the travel (i.e., the resources it currently occupies), d is the destination, and log is a history variable storing the path of resources the travel has occupied from its injection to its current location. The domain of all travels is denoted $\mathbb{T}$.

Note that a travel contains both meta data (e.g., an id and the log) and state data (e.g., the contents of the message and the location). In the ACL2 code, there is a clear distinction between both types of variables. It is ensured that the behavior of any network constituent is based only on the network state, and *not* on meta data. In this thesis, we do not distinguish between a message and its formal representation as a travel including meta data. We always use the term "message".

Given a list of messages T and an identifier id, let $T[id]$ return the message in T with identifier id. As identifiers are unique, there is always at most one such

message. If there is no such message, an error value is returned. We let T.ids denote the list of ids of the messages in T.

Configuration

A *configuration* σ is a tuple $\langle U, E, C\rangle$, where E denotes the list of enroute messages, i.e., messages that have been injected into the network, U is a list of uninjected messages, and C is a list of consumed messages, i.e, messages that have been en route but are removed from the network. The domain of all configurations is denoted Σ. Given a list of messages U, let $\sigma_\epsilon(U)$ denote the empty network configuration where U is the list of uninjected messages, formally, $\sigma_\epsilon(U) = \langle U, [], []\rangle$.

Again, note that a configuration contains both meta data (i.e., U and C) and state data (i.e., E). Any network constituent bases its behavior on the state only. For example, the routing function takes a configuration and yields a configuration. However, the routes may not be based on U or C, and additionally they cannot make use of, e.g., the logs of the enroute travels.

Given a configuration, function place can be used to retrieve the contents of a place in a resource. For example, place(r, n, σ) returns the contents of the nth place of resource r in configuration σ. An empty place is denoted by ϵ. If n is out of bounds, i.e., if n is greater than $|r|$, an error value is returned. Finally, generic function $\boldsymbol{\upsilon}(r, \sigma)$ returns true if and only if resource r is unavailable in configuration σ.

3.2.3 Generic Constituents

We informally describe the behavior of the generic constituents. Their exact behavior is axiomatized by the proof obligations presented in the next section.

Generic function $\mathbf{I} : \Sigma \times \mathbb{N} \mapsto \Sigma$ represents the *injection method*. Given a configuration and the current time, it decides which uninjected messages are ready for departure. It partitions list U of uninjected messages into a list D of departing messages and a list P of postponed messages. Departing messages are injected in the network and added to the list E of enroute ones. The list of departing messages is denoted $\mathbf{I}(\sigma, z).D$. The list of postponed messages is denoted $\mathbf{I}(\sigma, z).P$.

Generic function $\mathbf{R} : R \times R \mapsto \mathcal{P}(R)$ represents the routing function. From the current resource and a destination it computes a set of *next hops*. We overload this function to apply to a configuration. Function $\mathbf{R} : \Sigma \mapsto \Sigma$ loops over the set of enroute messages and applies function $\mathbf{R} : R \times R \mapsto \mathcal{P}(R)$ to each message. It stores the set of next hops as meta data in each enroute travel. The configuration yielded by function $\mathbf{R} : \Sigma \mapsto \Sigma$ is thus equivalent to its input, with for each enroute travel the next hops added as meta data. Note that function $\mathbf{R} : R \times R \mapsto \mathcal{P}(R)$ does not precompute an entire path for the message. It only supplies the next steps the message may take. After each move, the routing function is applied to the current location to recompute the set of next hops the message can take from that location.

Generic function $\mathbf{O} : \Sigma \mapsto \Sigma$ represents the *starvation prevention mechanism*. Starvation can be avoided using a correct resource assignment scheme [48]. Effectively, such a scheme tells the switching policy which messages have priority in case of contention. We represent a starvation prevention mechanism as a function

which reorders the list of enroute messages of a configuration, so that messages with high priority are placed at the start of the list of enroute messages.

Generic function $\mathbf{S} : \Sigma \mapsto \Sigma$ represents the *switching policy*. It performs two tasks. First, it moves enroute messages. To move a message, it checks whether there exists a next hop that is available and – in case of contention – whether it has priority over other messages. If so, the message will be moved from its current resource to the next. Secondly, it consumes messages that arrive at their destination. An arrived message is removed from the list of enroute messages $\sigma.E$ and added to the list of consumed messages $\sigma.C$.

3.2.4 Deadlock Configuration

A configuration is defined as a tuple of three lists. Not any tuple is a legal configuration. First, the list of enroute messages assigned to some resource may not exceed the capacity of that resource. This is expressed by function $\text{fits} : \Sigma \mapsto \mathbb{B}$. Secondly, the list of enroute messages may not violate invariants induced by the switching policy. For example, in a packet-switched network an induced invariant is that each message occupies at most one resource. Generic function $\boldsymbol{\iota}_{\mathbf{S}} : \Sigma \mapsto \mathbb{B}$ returns true if and only if the generic switching invariant holds for the given configuration.

Definition 3.1 Configuration σ is *legal*, notation $\text{legal}(\sigma)$, if and only if no capacities are exceeded and the generic switching invariant is satisfied.

$$\text{legal}(\sigma) \quad \stackrel{\text{def}}{=} \quad \text{fits}(\sigma) \wedge \boldsymbol{\iota}_{\mathbf{S}}(\sigma)$$

The definition of deadlock is dependent on the type of network. In some networks, deadlock can be defined as a global property where all messages are stuck. In some networks, deadlock is defined as a local property, where some message is permanently blocked. Generic predicate $\boldsymbol{\Omega} : \Sigma \mapsto \mathbb{B}$ returns true if and only if the given configuration σ is a legal deadlock configuration. Deadlock freedom is defined as the absence of deadlocks.

Definition 3.2 Network $\mathbf{N}$ is deadlock-free, notation $\mathsf{DLF}(\mathbf{N})$, if and only if there exists no deadlock configuration.

$$\mathsf{DLF}(\mathbf{N}) \equiv \forall \sigma \in \Sigma \cdot \neg\, \boldsymbol{\Omega}(\sigma)$$

3.2.5 The Behavior of the Generic Network

We now have the means to define function $\mathbf{N_{beh}}$ which defines the behavior of the generic network. Besides the current configuration, function $\mathbf{N_{beh}}$ takes as input the current time. Each recursive call corresponds to one step.

$$\mathbf{N_{beh}}(\sigma, z) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \sigma & \text{if} \quad \sigma.E = [] \wedge \sigma.U = [] \\ \sigma_{\mathbf{I}} & \text{if} \quad \boldsymbol{\Omega}(\sigma_{\mathbf{I}}) \wedge \sigma_{\mathbf{I}}.D = [] \\ \mathbf{N_{beh}}(\langle \sigma_{\mathbf{I}}.P, \sigma_{\mathbf{S}}.E, \sigma_{\mathbf{S}}.C \rangle, z+1) & \end{cases}$$

$$\begin{array}{lrcl} \text{where} & \sigma_{\mathbf{S}} & = & \mathbf{S}(\sigma_{\mathbf{O}}) \\ & \sigma_{\mathbf{O}} & = & \mathbf{O}(\sigma_{\mathbf{R}}) \\ & \sigma_{\mathbf{R}} & = & \mathbf{R}(\sigma_{\mathbf{I}}) \\ & \sigma_{\mathbf{I}} & = & \mathbf{I}(\sigma, z) \end{array}$$

First, function $\mathbf{N_{beh}}$ checks whether there are enroute or uninjected messages. If there are none, it terminates as all messages have been evacuated. If there are messages, function $\mathbf{N_{beh}}$ first applies the injection method to the current configuration σ, resulting in configuration $\sigma_{\mathbf{I}}$. Messages are injected. If the resulting configuration is a deadlock and there are no new injections, function $\mathbf{N_{beh}}$ terminates. Secondly, it applies the routing function to configuration $\sigma_{\mathbf{I}}$, resulting in configuration $\sigma_{\mathbf{R}}$. Routes are computed. Messages are reordered by a call to the starvation prevention mechanism $\mathbf{O}$, resulting in configuration $\sigma_{\mathbf{O}}$. Then function $\mathbf{N_{beh}}$ applies the switching policy to $\sigma_{\mathbf{O}}$. Each message advances by one hop if possible, yielding a new list of enroute message $\sigma_{\mathbf{S}}.E$. The switching policy may consume some messages, yielding a new list of consumed messages $\sigma_{\mathbf{S}}.C$. The new list of uninjected messages is the list of messages postponed by the injection method. Time is increased by 1. Function $\mathbf{N_{beh}}$ is called recursively with these new parameters.

Note that – without further proof obligations on the behavior of the generic constituents – this function does not necessarily terminate. For example, if the network contains a livelock, function $\mathbf{N_{beh}}$ executes the network perpetually. Proving termination is a major issue addressed in this chapter. In Section 3.4 we provide proof obligations which ensure termination.

In the next sections, we will reason about execution steps. We will denote one step of $\mathbf{N_{beh}}$ with $\mathbf{step}(\sigma, z)$, i.e.,:

$$\mathbf{step} = \mathbf{S} \circ \mathbf{O} \circ \mathbf{R} \circ \mathbf{I}$$

Table 3.1 summarizes the different functions and parameters of the GeNoC framework.

	Description
R	Set of resources R
$\|r\|$	Capacity of resource
$t = \langle id, org, msg, l, d, log \rangle$	Message t
$\mathbb{T}$	Domain of message lists
$T[id]$	The message with identifier id
T.ids	The list of ids in T
$\sigma = \langle U, E, C \rangle$	Configuration Uninjected, Enroute, Consumed
$\sigma_\epsilon(U)$	Empty network configuration $\langle U, [], [] \rangle$
fits(σ)	No resource capacities are exceeded
legal(σ)	Configuration σ is legal
Σ	Set of all configurations
place(r, n, σ)	The content of a place
ϵ	An empty place
step(σ, z)	One step
POι	Implicit proof obligations
DLF($\mathbf{N}$)	Deadlock freedom
$\mathbf{N}$	Generic communication network
RGen	Generic resource generator
$\boldsymbol{\upsilon}$	Generic unavailability predicate
I	Generic injection method
R	Generic routing function
S	Generic switching policy
O	Generic starvation prevention mechanism
$\boldsymbol{\iota}_\mathbf{S}$	Generic switching invariant
$\boldsymbol{\Omega}$	Generic deadlock configuration
$\mathbf{N}_\mathbf{beh}$	Generic network behavior

Table 3.1: *Notation and generic functions*

3.3 Functional Correctness

The previous section presented a formal model of communication networks. We proceed with formulating correctness properties for this model. This section considers functional correctness. We first define the correctness property. Then we formulate a set of proof obligations sufficient for functional correctness. Finally, we formulate and prove our functional correctness theorem.

3.3.1 Definition of Functional Correctness

Functional correctness consists of the following properties:

1. When a message is consumed, its current location is its destination.
2. All messages traverse a correct route through the network.
3. The content of messages does not change.

Before we formalize functional correctness, we define the notion of a correct route.

Definition 3.3 List $r = [r_0, r_1, \ldots r_n]$ is a *correct route* for destination d, notation correctroutep(r, d), if and only if it is path that can be established by the routing function.

$$\text{correctroutep}(r, d) \stackrel{\text{def}}{=} r_0 \in R \wedge \forall 0 < i \leq n \cdot r_i \in R \wedge r_i \in \mathbf{R}(r_{i-1}, d)$$

A correct route is a path of valid resources where each element is supplied as a next hop by the routing function from the previous element.

Definition 3.4 Network $\mathbf{N}$ is *functionally correct*, notation $\mathsf{FC}(\mathbf{N})$, if and only if:

$$\mathsf{FC}(N) \stackrel{\text{def}}{=} \forall U \subseteq \mathbb{T} \cdot \forall t \in \mathbf{N_{beh}}(\sigma_\epsilon(U), 0).C \cdot \begin{cases} \text{last}(t.\text{log}) = t.d \\ \text{correctroutep}(t.\text{log}, t.d) \\ t.\text{msg} = U[t.\text{id}].\text{msg} \end{cases}$$

The definition considers all consumed messages after termination of function $\mathbf{N_{beh}}$. It checks the logs and returns true only if the last element of each log corresponds to the desired destination of the message. The logs must constitute correct routes. Finally, if the content of the message is compared to its original content, these must be equal. A network is functionally correct if this holds for any list of uninjected messages U. We provide proof obligations sufficient for functional correctness. Functional correctness holds trivially if no message is consumed. A network that is stuck in a deadlock can still be functionally correct, as long as its route so far has been correct.

3.3.2 Proof Obligations for Functional Correctness

To ensure that messages traverse a correct route, the routing function must always supply at least one next hop. Also, no next hops may be supplied if a message has arrived at its destination.

Proof Obligation 1 Existence next hops
The routing function supplies a non-empty set of next hops as long as the message has not arrived at its destination.

$$\forall c, d \in R \cdot c \neq d \iff \mathbf{R}(c, d) \neq \emptyset$$

To prove functional correctness we need to know that messages do not suddenly appear in the network. The injection method can only inject messages from the list of uninjected messages. All messages must either be injected or postponed. Also, the set of injected messages must be disjunct from the postponed messages: a message cannot be duplicated. Similarly, the switching policy can only advance enroute messages. All enroute messages must either remain enroute or be consumed. The sets of enroute messages, consumed messages, and uninjected messages are always pairwise disjoint. This is expressed by the following proof obligations.

Proof Obligation 2 Partitioning injection method
The injection method partitions the uninjected messages into postponed and departing messages.

$$\{\sigma_{\mathbf{I}}.P, \sigma_{\mathbf{I}}.D\} \oplus \sigma.U \text{ where } \sigma_{\mathbf{I}} = \mathbf{I}(\sigma, z)$$

Proof Obligation 3 Partitioning switching policy
The switching policy partitions the enroute messages into enroute and consumed messages.

$$\{\sigma_{\mathbf{S}}.E, \sigma_{\mathbf{S}}.C\} \oplus \sigma.E \text{ where } \sigma_{\mathbf{S}} = \mathbf{S}(\sigma)$$

The switching policy can consume messages according to some consumption criterion. A restriction is that messages are consumed at their destination only, i.e., messages are not dropped at some intermediate location.

Proof Obligation 4 Consumption only at destinations
The switching policy consumes messages only at their destination.

$$\forall t \in \sigma_{\mathbf{S}}.C \cdot \text{last}(t.\text{log}) = t.d$$

Finally, both the injection method and the switching policy must always yield a legal configuration.

Proof Obligation 5 Legal configuration (injection)
Given a legal configuration, the injection method yields a legal configuration.

$$\text{legal}(\sigma) \quad \Longrightarrow \quad \text{legal}(\sigma_{\mathbf{I}}) \text{ where } \sigma_{\mathbf{I}} = \mathbf{I}(\sigma, z)$$

Proof Obligation 6 Legal configuration (switching)
Given a legal configuration, the switching policy yields a legal configuration.

$$\text{legal}(\sigma) \quad \Longrightarrow \quad \text{legal}(\sigma_{\mathbf{S}}) \text{ where } \sigma_{\mathbf{S}} = \mathbf{S}(\sigma)$$

3.3.3 Functional Correctness Theorem

Theorem 3.1 Proof Obligations 1 to 6 ensure functional correctness of generic network $\mathbf{N}$.

$$\mathrm{PO1}(\mathbf{N}) \text{ to } \mathrm{PO6}(\mathbf{N}), \mathrm{PO}\iota(\mathbf{N}) \models \mathsf{FC}(\mathbf{N})$$

Proof. The proof of functional correctness is by structural induction on function $\mathbf{N_{beh}}$. We do the proof for route correctness. The proofs that messages are consumed only at their destination and have unchanged content are similar.

An implicit proof obligation states that the logs for all uninjected messages are initially empty. The base case is therefore trivial. For the inductive case, we need to prove that if the logs contain correct routes, after one step the routes are still correct. We only need to consider the enroute messages, as we will prove that the logs of the uninjected messages are always empty and the logs of consumed messages remain unchanged.

Consider the call of the injection method in one step. New messages are injected and we must prove that these new enroute messages have correct routes. This is trivial, as the uninjected messages have empty logs. The calls of both the routing function and the starvation prevention mechanism do not alter the logs or the enroute messages and therefore preserve route correctness. It remains to be shown that the call of the switching policy preserves route correctness.

By construction of $\mathbf{N_{beh}}$, the switching policy takes as parameter a configuration where next hops have been computed by the routing function. By Proof Obligation 1 there exists at least one. If an enroute message moves, the switching policy adds the chosen next hop to the log. By typing of the routing function, it is a valid resource. The last element of the logs is the current position of the message. Adding the next hop as new element preserves the route correctness, as the new element is a next hop for the previous element and the rest of the route is correct by the induction hypothesis. This concludes the proof.

Two assumptions have been made here, which have yet to be proven. First, the logs of the uninjected messages are assumed to remain empty until injection. This holds since Proof Obligation 2 ensures that the list of uninjected messages is always disjunct from the departing messages. By induction, this implies that the list of uninjected messages is always disjunct from the enroute messages. Secondly, the logs of the consumed messages are assumed to remain unchanged. This holds, since Proof Obligation 3 ensures that the list of consumed messages is always disjunct from both the enroute and uninjected messages. □

Table 3.2 summarizes the notions and proof obligations related to functional correctness.

3.4 Evacuation

A network is *evacuatable* if and only if every uninjected message will eventually be injected and if every injected message will eventually be consumed. If, after termination of function $\mathbf{N_{beh}}$, the list of consumed messages is equal to the initial

	Description
correctroutep(r, d)	List of resources r is a valid route
$\mathsf{FC}(\mathbf{N})$	Functional Correctness
Proof Obligation 1	Existence next hops
Proof Obligation 2	Partitioning injection method
Proof Obligation 3	Partitioning switching policy
Proof Obligation 4	Consumption only at destinations
Proof Obligation 5	Legal configuration (injection)
Proof Obligation 6	Legal configuration (switching)

Table 3.2: *Notation and proof obligations related to functional correctness*

list of uninjected messages, all messages have been injected and all messages have been consumed.

Definition 3.5 Network $\mathbf{N}$ is *evacuatable*, notation $\mathsf{EVAC}(\mathbf{N})$, if and only if any list of uninjected messages U can be injected and processed by the network.

$$\mathsf{EVAC}(\mathbf{N}) \equiv \forall U \subseteq \mathbb{T} \cdot \mathbf{N}_{\mathbf{beh}}(\sigma_\epsilon(U), 0).C.\text{ids} \backsimeq U.\text{ids}$$

We define evacuation using the unique identifiers of the messages. After termination of function $\mathbf{N}_{\mathbf{beh}}$ the list of consumed messages must contain the same identifiers as the initial list of uninjected messages. This does not state anything about the actual contents of the messages. That is, given an identifier id of some consumed message m_c, our definition of evacuation does not require that this message is indeed equal to the initial uninjected message m_u with the same identifier. However, functional correctness ensures that identifiers indeed always refer to the same messages. Combined with functional correctness, we can prove that all messages correctly arrive at their destination.

3.4.1 Proof Obligations for Evacuation

If a network may reach a deadlock configuration, it is not evacuatable. We require deadlock freedom to prove evacuation. For now, deadlock freedom is not split up into proof obligations. Proof obligations sufficient to prove deadlock freedom is the major focus of this thesis. For sake of completeness, we formulate a proof obligation that simply requires the network to be deadlock-free.

Proof Obligation 7 Deadlock Freedom
The network is deadlock-free.

$$\mathsf{DLF}(\mathbf{N})$$

The remaining proof obligations for evacuation will be defined using two measures. The *injection measure*, denoted $\boldsymbol{\mu}_{\mathbf{I}}$, represents the progress of injection. Initially, when no message has been injected, the value of the injection measure is its maximum value. It decreases with each injection until all messages have been injected. The *switching measure*, denoted $\boldsymbol{\mu}_{\mathbf{S}}$ represents the progress of the enroute messages. It decreases with the progression of each enroute message. It may increase when new messages are injected.

Injection Measure

Function $\boldsymbol{\mu_I} : \mathbb{T} \times \mathbb{N} \mapsto \mathbb{N}$ takes as parameters the list of uninjected messages and the current time. It returns the injection measure. We define the following proof obligations:

Proof Obligation 8 Liveness of injection
Given an empty network and a non-empty list of uninjected messages, the injection measure decreases after one call of the injection method.

$$\sigma.E = [] \wedge \sigma.U \neq [] \implies \boldsymbol{\mu_I}(\sigma_{\mathbf{I}}.P, z+1) < \boldsymbol{\mu_I}(\sigma.U, z) \quad \text{where } \sigma_{\mathbf{I}} = \mathbf{I}(\sigma, z)$$

Note that the injection measure only needs to decrease in case of an empty network. This proof obligation effectively formulates liveness: if nothing has been injected yet, at some point injection must start.

At each step where the network is empty, the injection measure decreases. If in other steps it increases, the measure cannot be used to prove termination of injection. An infinite injection sequence can exist where the measure decreases and increases infinitely often. We therefore need to ensure that the injection measure never increases.

Proof Obligation 9 Partial ordering injection measure
The injection measure never increases.

$$\boldsymbol{\mu_I}(\sigma_{\mathbf{I}}.P, z+1) \leq \boldsymbol{\mu_I}(\sigma.U, z) \quad \text{where } \sigma_{\mathbf{I}} = \mathbf{I}(\sigma, z)$$

Proof Obligation 10 Correctness injection measure
If after injection the injection measure has not changed, then no injections have occurred.

$$\boldsymbol{\mu_I}(\sigma_{\mathbf{I}}.P, z+1) = \boldsymbol{\mu_I}(\sigma.U, z) \implies \sigma_{\mathbf{I}} = \sigma \quad \text{where } \sigma_{\mathbf{I}} = \mathbf{I}(\sigma, z)$$

Proof obligation 10 formulates correctness of the injection measure: if the injection method injects new messages into the network, the injection measure decreases.

Combined, Proof Obligations 8, 9, and 10 are sufficient to prove termination of injection. In each step, injection method $\mathbf{I}$ is called. As long as the network is empty, the injection measure decreases. At some point injection must start. With each consecutive injection the injection measure decreases. Since it never increases, the sequence of injections terminates.

Switching Measure

Function $\boldsymbol{\mu_S} : \Sigma \mapsto \mathcal{L}(\mathbb{N})$ takes as parameter the current configuration. It returns the switching measure, which is a list of natural numbers. We define the following proof obligation:

Proof Obligation 11 Livelock freedom
If there is no deadlock and there are enroute messages, the switching measure decreases under lexicographical ordering after one call of switching and routing.

$$\neg\, \mathbf{\Omega}(\sigma_{\mathbf{R}}) \wedge \sigma_{\mathbf{R}}.E \neq [] \implies \boldsymbol{\mu}_{\mathbf{S}}(\mathbf{S}(\sigma_{\mathbf{R}})) < \boldsymbol{\mu}_{\mathbf{S}}(\sigma) \quad \text{where } \sigma_{\mathbf{R}} = \mathbf{R}(\sigma)$$

This proof obligation ensures livelock freedom. If the network is not in a deadlock, at least one message can progress towards its destination. There must exist a measure which decreases with each step where the switching policy is applied to a deadlock-free configuration. This excludes livelock, as messages cannot infinitely move around in the network.

The switching measure represents the progress of the enroute messages. Reordering the list of enroute messages does not influence the progress of these messages. The switching measure must remain equal if the list is reordered. This is expressed as a proof obligation.

Proof Obligation 12 Irrelevance ordering enroute messages
The ordering of the messages passed to the switching measure is irrelevant for the result.

$$\sigma_0.E \backsimeq \sigma_1.E \implies \boldsymbol{\mu}_{\mathbf{S}}(\sigma_0) = \boldsymbol{\mu}_{\mathbf{S}}(\sigma_1)$$

Finally, we need to know that the starvation prevention mechanism reorders the list of messages in its configuration.

Proof Obligation 13 Reordering starvation prevention mechanism
The starvation prevention mechanism reorders the list of enroute messages.

$$\mathbf{O}(\sigma).U = \sigma.U \wedge \mathbf{O}(\sigma).E \backsimeq \sigma.E \wedge \mathbf{O}(\sigma).C = \sigma.C$$

3.4.2 Evacuation Theorem

Theorem 3.2 Proof Obligations 7 to 13 ensure generic network $\mathbf{N}$ is evacuatable if and only if it is deadlock-free.

$$\text{PO8}(\mathbf{N}) \text{ to } \text{PO13}(\mathbf{N}), \text{PO}\iota(\mathbf{N}) \models \mathsf{EVAC}(\mathbf{N})$$

Proof. Function $\mathbf{N}_{\mathbf{beh}}$ terminates in two cases. In the first case, all initially uninjected messages have been injected into the network and have been consumed. In this case $\mathbf{N}_{\mathbf{beh}}(\sigma, z).C.\text{ids} \backsimeq \sigma.U.\text{ids}$. In the second case, a deadlock configuration has occurred. In this case $\mathbf{N}_{\mathbf{beh}}(\sigma, z).C.\text{ids} \not\backsimeq \sigma.U.\text{ids}$. By Proof Obligation 7, the second case cannot occur. Thus a network is evacuatable if function $\mathbf{N}_{\mathbf{beh}}$ terminates.

Termination is proven using a termination measure, i.e., a measure that decreases under some well-founded relation. The termination measure of function $\mathbf{N}_{\mathbf{beh}}$ is defined as follows:

$$\boldsymbol{\mu}_{\mathbf{N}}(\sigma, t) \stackrel{\text{def}}{=} [\boldsymbol{\mu}_{\mathbf{I}}(\sigma.U, t)] \sqcup \boldsymbol{\mu}_{\mathbf{S}}(\sigma)$$

The termination measure is a list of two elements: first the injection measure and second the switching measure. It decreases with each recursive call of $\mathbf{N_{beh}}$ under lexicographical ordering.

In one recursive call function $\mathbf{N_{beh}}$ applies injection, routing, reordering, and switching. The proof proceeds by case distinction: the injection measure either increases, decreases, or stays equal. By Proof Obligation 9 the first case cannot occur. In the second case, the termination measure decreases under lexicographical ordering, regardless of the switching measure. It remains to be shown that in the third case the switching measure decreases.

Assume that the injection measure stays equal. By Proof Obligation 10 there are no new injections. We can ignore the injection method in this recursive call of function $\mathbf{N_{beh}}$. By Proof Obligations 12 and 13, the call of starvation mechanism $\mathbf{O}$ has no influence on the switching measure. Thus Proof Obligation 11 can be applied: the switching measure decreases after one call of switching and routing. This completes the proof, as measure $\boldsymbol{\mu}_{\mathbf{N}}$ decreases. However, Proof Obligation 11 has two assumptions which have to be discharged: the current configuration is not a deadlock configuration and there are enroute messages. If the first assumption does not hold then there is a deadlock configuration. Since by Proof Obligation 10 there are no departing messages, the second termination condition of function $\mathbf{N_{beh}}$ is satisfied and it terminates. If the second assumption does not hold then there are no enroute messages. The network is empty. By Proof Obligation 8 the injection measure decreases, which contradicts the assumption that the injection measure stays equal.

From Proof Obligations 8 to 13 it follows that function $\mathbf{N_{beh}}$ terminates. As by Proof Obligation 7 there does not exist a deadlock, $\mathbf{N_{beh}}(\sigma_\epsilon(U), 0).C.\text{ids} \backsimeq U.\text{ids}$ for any list of uninjected messages U. □

Note that the return values of the injection and the switching measure are restricted to respectively a natural number and a list of natural numbers. The corresponding well-founded relations are respectively $<$ and the lexicographical ordering. From these values, it is easy to construct the termination measure $\boldsymbol{\mu}_{\mathbf{N}}$ and the corresponding well-founded relation. Ideally, the proof obligations would allow the values of the measures to decrease under arbitrary well-founded relations.

Theorem 3.2 states that evacuation follows from deadlock freedom, livelock freedom, and liveness. Table 3.3 summarizes the different notions and proof obligations related to deadlock freedom and evacuation.

3.5 Local Liveness

We formalize local liveness with another measure. We call this measure the *waiting measure*, denoted $\boldsymbol{\mu}_{\texttt{wait}}$. The waiting measure takes as parameters a message t and the current configuration. It represents the maximum time it takes for t to wait in its current location, i.e., the maximum time t can be blocked.

Definition 3.6 Network $\mathbf{N}$ is *locally live*, notation $\mathsf{LL}(\mathbf{N})$, if and only if:

$$\begin{aligned}\mathsf{LL}(\mathbf{N}) \quad \equiv \quad & \forall \sigma \in \Sigma \cdot \forall t \in \sigma.E \cdot \forall z \in \mathbb{N} \cdot \\ & \mathbf{step}(\sigma, z).E[t.id] = t \implies \boldsymbol{\mu}_{\texttt{wait}}(t, \mathbf{step}(\sigma, z)) < \boldsymbol{\mu}_{\texttt{wait}}(t, \sigma)\end{aligned}$$

	Description
$\mathsf{EVAC}(\mathbf{N})$	Network $\mathbf{N}$ is evacuatable
$\boldsymbol{\mu}_{\mathbf{I}}$	Generic injection measure
$\boldsymbol{\mu}_{\mathbf{S}}$	Generic switching measure
Proof Obligation 7	Deadlock freedom
Proof Obligation 8	Liveness
Proof Obligation 9	Partial ordering $\boldsymbol{\mu}_{\mathbf{I}}$
Proof Obligation 10	Correctness $\boldsymbol{\mu}_{\mathbf{I}}$
Proof Obligation 11	Livelock freedom
Proof Obligation 12	Irrelevance ordering enroute messages
Proof Obligation 13	Reordering starvation prevention

Table 3.3: *Notation, generic functions, and proof obligations related to evacuation*

Notation $\mathbf{step}(\sigma, z).E[t.id] = t$ expresses the fact that message t does not move during one step. The message is enroute and is not modified. Measure $\boldsymbol{\mu}_{\texttt{wait}}$ decreases with each step in which message t does not move. If measure $\boldsymbol{\mu}_{\texttt{wait}}$ can be defined in any configuration, for any enroute message, the time a message remains in one resource is always finite.

3.5.1 Proof Obligations for Local Liveness

Local liveness requires absence of deadlocks. We need an additional proof obligation to enforce that messages move when possible. This excludes, e.g., a switching policy that delays messages arbitrarily. The proof obligation states that if there is a message that is permanently blocked with unavailable next hops, there exists a deadlock configuration. Local liveness also requires starvation freedom. We define two proof obligations which are sufficient for starvation freedom. In combination with Proof Obligation 7, these three proof obligations are sufficient to prove the existence of the measure $\boldsymbol{\mu}_{\texttt{wait}}$, and thereby the absence of permanent blocking.

Proof Obligation 14 states that if a network is deadlock-free, it possible to define a measure called the *deadlock measure*. Function $\boldsymbol{\mu}_{\boldsymbol{\Omega}} : \mathbb{T} \times \Sigma \mapsto \mathbb{N}$ returns this measure. It takes as parameters a message t and the current configuration. This measure represents the maximum time that message t can wait for unavailable resources. It decreases with each step where all next hops are unavailable.

Proof Obligation 14 Deadlock measure
Deadlock freedom implies that messages are not permanently blocked due to unavailable next hops.

$$\begin{aligned}\mathsf{DLF}(\mathbf{N}) \implies\ & \forall \sigma \in \Sigma \cdot \forall t \in \sigma.E \cdot \forall z \in \mathbb{N} \cdot \\ & (\forall r \in \mathbf{R}(t.l, t.d) \cdot \boldsymbol{\upsilon}(r, \sigma)) \implies \boldsymbol{\mu}_{\boldsymbol{\Omega}}(t, \mathbf{step}(\sigma, z)) < \boldsymbol{\mu}_{\boldsymbol{\Omega}}(t, \sigma)\end{aligned}$$

The proof obligations sufficient for starvation freedom use another measure called the *starvation measure*, denoted $\boldsymbol{\mu}_{\mathbf{O}}$. Starvation occurs when a message has to wait infinitely many times to acquire a resource, even though this resource becomes available from time to time. If the time a message has to wait for an *available* resource is bounded, no starvation can occur. The starvation measure

represents the maximum time a message has to wait for an available resource. If such a measure can be defined, the number of times a message is not granted a required available resource is finite.

Function $\boldsymbol{\mu_O} : \mathbb{T} \times R \times \Sigma \mapsto \mathbb{N}$ represents the starvation measure. It takes as parameters a message t, a next hop of message t, and the current configuration. We define the following proof obligation:

Proof Obligation 15 Starvation Freedom
The time a message is not granted an available resource is finite.

$$\forall t \in \sigma.E \cdot \forall r \in \mathbf{R}(t.l, t.d) \cdot \quad \neg\, \boldsymbol{\upsilon}(r, \sigma) \wedge \mathbf{step}(\sigma, z).E[t.id] = t \implies \boldsymbol{\mu_O}(t, r, \mathbf{step}(\sigma, z)) < \boldsymbol{\mu_O}(t, r, \sigma)$$

For any enroute message t and for any next hop r, if the next hop is available and the message does not move, then the starvation measure decreases. This implies that resource r is not granted to other messages infinitely often. Here, we need to ensure that the starvation measure never increases.

Proof Obligation 16 Partial ordering starvation measure
The starvation measure never increases.

$$\forall t \in \sigma.E \cdot \forall r \in \mathbf{R}(t.l, t.d) \cdot \quad \mathbf{step}(\sigma, z).E[t.id] = t \implies \boldsymbol{\mu_O}(t, r, \mathbf{step}(\sigma, z)) \leq \boldsymbol{\mu_O}(t, r, \sigma)$$

For any enroute message and for any next hop r, as long as message t does not move, the starvation measure does not increase.

Defining a waiting measure is a hard problem, as it requires defining worst-case bounds on latency of messages in the network. Defining a deadlock measure is equally hard. It is however relatively easy to prove the contrapositive version of Proof Obligation 14, i.e., that if a message is permanently blocked, there exists a deadlock. Such a proof merely shows the existence of a deadlock measure for a deadlock-free network. The starvation measure is easy to define. Because we do not define the deadlock measure, but merely show its existence, we also do not define the waiting measure, but just show its existence.

3.5.2 Local Liveness Theorem

We now prove local liveness from the proof obligations.

Theorem 3.3 Proof Obligations 7 and 14 to 16 ensure local liveness of generic network $\mathbf{N}$.

$$\mathrm{PO7}(\mathbf{N}), \mathrm{PO14}(\mathbf{N}) \text{ to } \mathrm{PO16}(\mathbf{N}), \mathrm{PO}\iota(\mathbf{N}) \models \mathsf{LL}(\mathbf{N})$$

Proof. Let t be an enroute message in configuration σ waiting for next hop r. We prove the existence of waiting measure $\boldsymbol{\mu}_{\texttt{wait}}$.

Proof Obligation 7 ensures network N is deadlock-free. By Proof Obligation 14, this means that no message can be infinitely blocked by *unavailable* next hops. From the deadlock measure $\boldsymbol{\mu_\Omega}$ and the starvation measure $\boldsymbol{\mu_O}$ we construct $\boldsymbol{\mu}_{\texttt{wait}}$. Message t waits to acquire a next hop. It takes at most $\boldsymbol{\mu_\Omega}(t, \sigma)$ steps before a

next hop becomes available. When this happens, either message t acquires it or not. By Proof Obligations 15 and 16, the second case happens at most $\boldsymbol{\mu_O}(t, r, \sigma)$ times per resource r. Figure 3.2 graphically depicts this process.

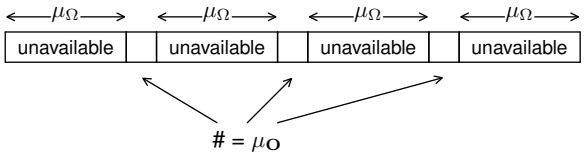


Figure 3.2: *Waiting measure*

Thus the waiting measure can be defined as follows:

$$\boldsymbol{\mu}_{\mathtt{wait}}(t, \sigma) \stackrel{\text{def}}{=} \boldsymbol{\mu_\Omega}(t, \sigma) \cdot \sum_{\forall r \in \mathbf{R}(t.l, t.d)} \boldsymbol{\mu_O}(t, r, \sigma)$$

□

Table 3.4 summarizes the different notions and proof obligations related to local liveness.

	Description
$\mathsf{LL}(\mathbf{N})$	Local Liveness
$\boldsymbol{\mu}_{\mathtt{wait}}$	Generic waiting measure
$\boldsymbol{\mu_\Omega}$	Generic deadlock measure
$\boldsymbol{\mu_O}$	Generic starvation measure
Proof Obligation 14	Deadlock measure
Proof Obligation 15	Starvation freedom
Proof Obligation 16	Partial ordering $\boldsymbol{\mu_O}$

Table 3.4: *Notation, generic functions, and proof obligations related to local liveness.*

3.6 Productivity

We can now formally define productivity.

Definition 3.7 Network $\mathbf{N}$ is *productive*, notation $\mathsf{PROD}(\mathbf{N})$, if and only if any list of uninjected messages U can be injected and correctly processed by the network.

$$\mathsf{PROD}(\mathbf{N}) \equiv \mathsf{FC}(\mathbf{N}) \wedge \mathsf{EVAC}(\mathbf{N}) \wedge \mathsf{LL}(\mathbf{N})$$

The final theorem of correctness is a direct corollary of the theorems in this chapter.

Corollary 3.1 Proof Obligations 1 to 16 ensure productivity of generic network $\mathbf{N}$.

$$\mathrm{PO1}(\mathbf{N}) \text{ to } \mathrm{PO16}(\mathbf{N}), \mathrm{PO}\iota(\mathbf{N}) \models \mathsf{PROD}(\mathbf{N})$$

Table 3.5 gives an overview of all generic functions. To make an instantiation of a complete network, one needs to provide a definition of all these functions. From these definitions, an executable specification of the network is automatically generated.

Function	Description
RGen	Resource generator
I	Injection method
R	Routing function
S	Switching policy
O	Starvation prevention mechanism
Ω	Deadlock configuration
$\iota_{\mathbf{S}}$	Switching invariant
υ	Unavailability predicate
$\mu_{\mathbf{I}}$	Injection measure
$\mu_{\mathbf{S}}$	Switching measure
$\mu_{\mathbf{O}}$	Starvation measure

Table 3.5: *Generic functions*

Table 3.6 gives an overview of all proof obligations. Proof Obligations 1 to 6 are sufficient to prove functional correctness. Proof obligations 7 states deadlock freedom. Proof obligations 8 to 13 are sufficient for evacuation. Finally, Proof Obligations 14 to 16 formulate starvation freedom. The third column shows the generic constituents on which the proof obligations depend.

PO	Name	Generics
1	Existence next hops	**R**
2	Partitioning injection method	**I**
3	Partitioning switching method	**S**
4	Consumption only at destinations	**S**
5	Legal configuration (injection)	**I**
6	Legal configuration (switching)	**S**
7	Deadlock freedom	**I**, **R**, **O**, **S**
8	Liveness	**I**
9	Partial ordering injection measure	**I**
10	Correctness injection measure	**I**
11	Livelock freedom	**R**, **S**
12	Irrelevance ordering enroute messages	**R**, **S**
13	Reordering starvation prevention mechanism	**O**
14	Deadlock measure	**Ω**, **S**
15	Starvation freedom	**O**
16	Partial ordering $\mu_{\mathbf{O}}$	**O**

Table 3.6: *Proof Obligations*

Figure 3.3 gives an overview of the proof structure. In order to prove productivity, one first needs to prove functional correctness. This amounts to discharging

Proof Obligations 1 to 6. Evacuation requires a proof of deadlock freedom and termination. Proof Obligations 7 to 13 are sufficient for this. Local liveness requires a proof of deadlock freedom and the discharging of the three additional proof obligations 14 to 16 to prove starvation freedom. The proofs of evacuation and functional correctness do not depend on these three constraints.

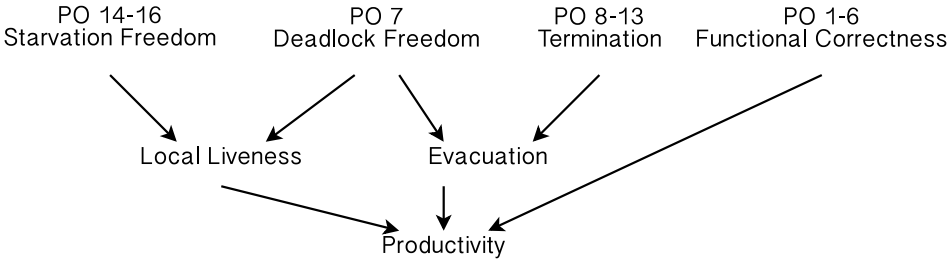


Figure 3.3: *Overview of the proof structure*

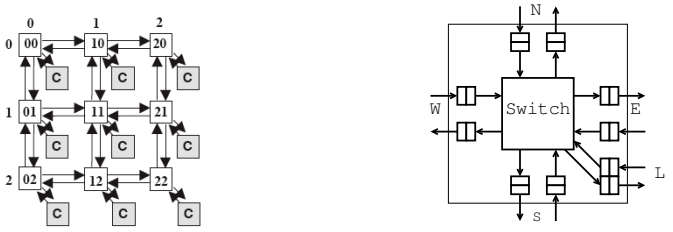
Our methodology requires an instantiation of 11 generic functions and a proof of 16 proof obligations. The next chapter demonstrates our methodology on a realistic example of a Network-on-Chip.

Chapter 4
Application to HERMES

The previous chapter extended the GeNoC framework for productivity proofs. A formal model of communication networks has been introduced, together with 16 proof obligations sufficient for productivity. The methodology is valuable as long as discharging the proof obligations is significantly easier than proving productivity from scratch. This chapter demonstrates our methodology on an academic example of a NoC called HERMES [99]. We instantiate all generic functions and discharge all proof obligations. An overview is presented of the proof effort required to prove productivity of HERMES using our methodology.

4.1 HERMES

HERMES is based on a 2D mesh architecture (Figure 4.1a). Each core is connected to a processing node. Each processing node contains a switch connected to five bi-directional ports. Ports East, West, North, and South connect to the neighbor switches. Port Local connects to the core (Figure 4.1b). The resources of the network are the ports.



(a) *3x3 HERMES mesh* **(b)** *Node with two places per port*

Figure 4.1: *A 2D-Mesh HERMES NoC*

We specify a HERMES chip with *west-first* routing and wormhole switching [61]. West-first routing is an adaptive routing function for a 2D mesh which first routes west and then adaptively south, east, and north. The principle of

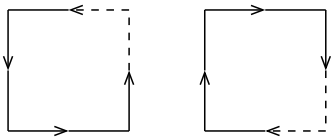


Figure 4.2: *Turns in west-first routing. Dashed turns are prohibited.*

west-first routing is that all turns to the west are restricted. This breaks many dependency cycles (see Figure 4.2).

Restricting all turns to the west does not break all cycles. There may be a cycle if one first routes north, turns from north to south, and then from south to north again. One of these turns must be prohibited as well. Arbitrarily, we have chosen to restrict the south-north turn. This breaks all dependency cycles and consequently there is no deadlock.

Even though the principle behind west-first routing is straightforward, there are subtleties in defining it formally. First, in some ports the set of possible destinations of the messages located in the port is restricted, e.g., a message in an eastern out-port cannot be destined for any destination west of that port, as this would mean that the message would have to turn west after going east.

Secondly, routing is *not* memoryless. That is, routing does not solely depend on the current processing node and the destination. West-first routing needs to be defined from channel to channel, instead of from processing node to channel. When a message arrives at a processing node, to be forwarded to another, one step of its history is needed to make the routing decision: from which direction the message entered the processing node. For example, whether a message can be routed west depends on at which in-port the message had arrived, i.e., from which channel the message came. Only if a message arrives at the eastern or local in-port of the processing node can it be routed west.

The cores generate and consume messages. Injection occurs in the local in-ports of the processing nodes. Consumption occurs at local out-ports. We assume that once a message arrives at the local out-port of the destination it is consumed by the core. The GeNoC framework models the behavior of the cores, i.e., message generation, by assuming an arbitrary and unbounded – but finite – list of uninjected messages. With each message, an arbitrary *desired injection time* is associated, representing the time at which the core generates the message. After generation of the message it is injected into the local in-port as soon as all places of this port are empty.

Messages are handled in a first-in-first-out (FIFO) order. During the time a message m waits to acquire a port p, new messages that want to acquire port p are put back in line and therefore served after message m. This prevents starvation, as an available port cannot be granted to other messages infinitely often.

4.2 User Input, Part I: Executable Specification

In this section we provide instantiated functions for each generic function of the GeNoC framework (see Table 3.5). An instantiation `i` of a generic function **F** is

denoted $\mathbf{F}^{\mathtt{i}}$.

Resource generator

Each port p is represented as a tuple $\langle x, y, c, d\rangle$ where x and y specify the coordinates of the processing node to which p belongs, c is the channel name of the port (`N`, `S`, `E`, `W` or `L`) and d is the direction (`I` or `O`). The resource generator takes a single parameter dim which represents the dimension of the mesh. Function $\mathbf{RGen}^{\mathtt{2D}}$ generates up to ten ports for each processing node (x, y) such that $x < dim.x$ and $y < dim.y$. Some ports on the border of the mesh are never used. For example, both the northern and western ports of processing node $(0, 0)$ are useless. We let the resource generator remove these border ports from the set of resources.

Switching Policy

We reuse the specification of the wormhole switching policy provided by the GeNoC framework [21]. Let $\mathbf{S}^{\mathtt{whs}}$ be that function. For each message, function $\mathbf{S}^{\mathtt{whs}}$ checks whether there exists an available next hop. If so, the message advances by one hop. If it arrives at its destination, it is removed from the configuration and added to the list of consumed messages.

A port accepts a header flit if it is completely empty. Thus predicate $\boldsymbol{\upsilon}^{\mathtt{whs}}$ is defined as follows:

$$\boldsymbol{\upsilon}^{\mathtt{whs}}(p, \sigma) \stackrel{\text{def}}{=} \exists 0 \leq i < |p| \cdot \text{place}(p, i, \sigma) \neq \epsilon$$

A deadlock is a configuration in which all enroute messages are blocked. A message is blocked if and only if all its next hops are unavailable.

$$\boldsymbol{\Omega}^{\mathtt{whs}}(\sigma) \stackrel{\text{def}}{=} |\sigma.E| > 0 \wedge \forall t \in \sigma.E \cdot \forall r \in \mathbf{R}(t.l, t.d) \cdot \boldsymbol{\upsilon}(r, \sigma)$$

Injection Method

We assume that a desired injection time is associated with each message, denoted $t.z$. Function $\mathbf{I}^{\mathtt{z}}$ injects messages as soon as possible after their desired injection time.

Let function $\text{put}(id, r, \sigma)$ place id in an empty buffer of resource r in configuration σ. Let function $\text{postpone}(t, \sigma)$ add travel t to the list of uninjected travels $\sigma.U$. Function $\mathbf{I}^{\mathtt{z}}$ checks for all messages whether its desired injection time has been reached and whether the origin port of the message is available. If so, the message is injected. Otherwise injection of the message is delayed. The instantiation of the injection method uses recursive function $\mathbf{I}^{\mathtt{z}}_{\text{rec}}$, which is defined as follows.

$$\mathbf{I}^{\mathtt{z}}_{\text{rec}}(\sigma, z, n) \stackrel{\text{def}}{=} \begin{cases} \sigma & \text{if } n >= |\sigma.U| \\ \mathbf{I}^{\mathtt{z}}_{\text{rec}}(\text{put}(u_n.id, u_n.org, \sigma), z, n+1) & \text{if } \begin{cases} \neg\, \boldsymbol{\upsilon}^{\mathtt{whs}}(u_n.org, \sigma) \\ u_n.z \leq z \end{cases} \\ \mathbf{I}^{\mathtt{z}}_{\text{rec}}(\text{postpone}(u_n, \sigma), z, n+1) & \text{if otherwise} \end{cases}$$
$$\text{where } u_n = \sigma.U[n]$$

Injection method $\mathbf{I}^{\mathtt{z}}$ is defined as:

$$\mathbf{I}^{\mathtt{z}}(\sigma, z) \stackrel{\text{def}}{=} \mathbf{I}^{\mathtt{z}}_{\text{rec}}(\sigma, z, 0)$$

Routing Function

Function next_in(p) returns the in-port connected to out-port p. For example, the in-port connected to the eastern out-port of processing node $(0, 0)$ is the western in-port of processing node $(1, 0)$:

$$\text{next\_in}(\langle 0, 0, \mathtt{E}, \mathtt{O}\rangle) = \langle 1, 0, \mathtt{W}, \mathtt{I}\rangle$$

Function next_in is not defined for local out-ports. Function trans($p, \mathtt{P}, \mathtt{D}$) returns the port specified by P and D in the same processing node as p:

$$\text{trans}(p, \mathtt{P}, \mathtt{D}) \stackrel{\text{def}}{=} \langle p.x, p.y, \mathtt{P}, \mathtt{D}\rangle$$

Routing function $\mathbf{R}^{\mathtt{wf}}$ computes a set of next ports from the current port, a destination and the dimension of the mesh. Routing depends on the dimension of the mesh as the routing logic is not equal in all processing nodes. For example, nodes in the rightmost column cannot route east, whereas all other nodes can always route east.

$$\mathbf{R}^{\mathtt{wf}}(p, d, dim) \stackrel{\text{def}}{=} \begin{cases} \text{next\_in}(p) & \text{iff} \quad p.d = \mathtt{O} \\ \text{trans}(p, \mathtt{W}, \mathtt{O}) & \text{iff} \quad p.c \in \{\mathtt{E}, \mathtt{L}\} \wedge p.x > 0 \\ \text{trans}(p, \mathtt{E}, \mathtt{O}) & \text{iff} \quad d.x > p.x \wedge p.x < dim.x \\ \text{trans}(p, \mathtt{N}, \mathtt{O}) & \text{iff} \quad d.x \geq p.x \wedge p.y > 0 \wedge p.c \neq \mathtt{N} \\ \text{trans}(p, \mathtt{S}, \mathtt{O}) & \text{iff} \quad \begin{cases} (d.x > p.x \vee (d.x = p.x \wedge d.y > p.y)) \\ p.y < dim.y \end{cases} \\ \text{trans}(p, \mathtt{L}, \mathtt{O}) & \text{iff} \quad d.x = p.x \wedge d.y = p.y \end{cases}$$

We provide detailed information on each case distinction:

- If the current port is an out-port, simply route to the connected in-port.
- It is only possible to route west if the message has just been injected – the current port is local-in – or if the message was already going west – the current port is east-in.
- It is possible to route east as long as the destination is east and as long as it does not lead out of the dimension of the network. Going east if the destination is not east, would mean that at some point we would have to route west again, resulting in a violation of the west-first principle.
- It is possible to route north as long as we are either in the right column or if the destination is east of us. Routing north if the destination is west would mean that after going north we would have to route west, resulting in a violation of the west-first principle. Furthermore, we can only route north if it does not lead outside of the dimension. Lastly, we prevent south-north turns. We cannot go north if we have been going in southern direction, i.e., if we are in a northern in-port.

- It is always possible to route south if the destination is east of us. If we are already in the right column, it is only possible to route south if the destination is south. Otherwise, a south-north turn would be required to reach the destination, whereas such turns are prohibited.

- Lastly, it is only possible to route to the local out-port if we have reached the destination.

Starvation prevention method

Function $\mathbf{O}^{\mathtt{F}} : \Sigma \mapsto \Sigma$ represents a FIFO starvation prevention mechanism. It sorts the list of messages by pushing all messages that have moved in the latest step to the back of the list of enroute messages. We let switching function $\mathbf{S}^{\mathtt{whs}}$ keep track of whether messages move or not. Function starving(t) returns true if and only if message t has not moved in the last call of $\mathbf{S}^{\mathtt{whs}}$. We define function $\mathbf{O}^{\mathtt{F}}$ as follows:

$$\mathbf{O}^{\mathtt{F}}(\sigma) \stackrel{\text{def}}{=} \langle \sigma.U, [t \in \sigma.E \mid \text{starving}(t)] \sqcup [t \in \sigma.E \mid \neg\, \text{starving}(t)], \sigma.C \rangle$$

4.3 User Input, Part II: Proofs

There are sixteen proof obligations to discharge. Table 4.1 gives an overview of these proof obligations and the effort required to discharge them. Six proof obligations are automatically discharged by ACL2 and no effort is required. Proof Obligations 3, 4, 6, and 14 depend on the switching policy only. They have been discharged in the wormhole switching module. Proof Obligation 5 could be proven completely with theorems from the wormhole switching module. Only five proof obligations require interaction with the ACL2 system. Two of them are straightforward. The final three are more difficult. They are related to the measures and to deadlock freedom. These two aspects are detailed hereafter.

The complete proof presented in Part II, including the generic model, the proofs of all the generic theorems, the specification of the HERMES NoC in the ACL2 logic, and discharging of all proof obligations consists of 12258 lines of ACL2 code. Only 1741 lines of code deal with the HERMES specific topology, routing function, the injection method and the discharging of the corresponding proof obligations. We will now provide details on the proofs.

4.3.1 Discharging Proof Obligations

The most intricate part of discharging the proof obligations is defining correct measures. We define the injection, switching and starvation measures and prove them correct. For each measure, we first provide the formal definition of the measure and then formulate the major assisting lemma's that were required to be proven in ACL2. We prove informally that there exists a deadlock measure.

PO	Lines	Aux. Theorems	Proof Effort (hrs)
1	6	0	0
2	142	27	1
3	N/A	N/A	N/A
4	N/A	N/A	N/A
5	N/A	N/A	N/A
6	N/A	N/A	N/A
7	834	70	12
8	11	0	0
9	5	0	0
10	10	0	0
11	307	18	4
12	17	5	0.5
13	0	0	0
14	N/A	N/A	N/A
15	367	38	8
16	63	0	0

Table 4.1: *Proof effort*

Injection measure: Definition

We need to define a measure that decreases with each injection and if the network is empty. We define function $\boldsymbol{\mu}_{\mathbf{I}}^{\mathbf{z}}$ as follows:

$$\boldsymbol{\mu}_{\mathbf{I}}^{\mathbf{z}}(U, z) \stackrel{\text{def}}{=} \sum_{t \in U} (1 + |t.z - z|)$$

The injection measure adds at least a value of one for each uninjected message. If the message is uninjected and its desired injection time has not been reached, the difference between the desired injection time and the current time is added, i.e., the time the message has to wait before its desired injection time has been reached.

With each injection this measure decreases, as each message adds at least 1 to the measure. If the network is empty, there are two cases: either there exists a message whose injection time has been reached or not. In the first case, at least one message will be injected and the measure decreases. In the second case, the time the messages have to wait decreases, i.e., for each message t the value of $|t.z - z|$ decreases with one. If the network is empty, the measure decreases with each call of $\mathbf{I}^{\mathbf{z}}$, regardless of whether there are injections or not.

Injection measure: Proof

Proof obligations 8 to 10 required no further proof effort, i.e., no further auxiliary theorems, hints, or interactive theorem proving, to be discharged.

Switching measure: Definition

For the switching measure we consider the *maximum route*, i.e., the longest route that can be taken from the current port to *some* destination. As Figure 4.3 shows, the maximum route is always the route destined for the upper-right processing node. Regardless of the actual destination, the length of this maximum route always decreases when progressing towards the actual destination, because the actual destination is always somewhere on the maximum route.

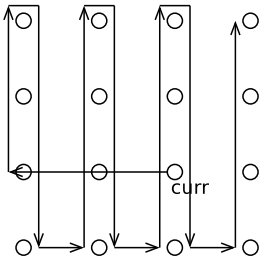


Figure 4.3: *The maximum west-first route from* `curr`

The maximum route is taken if a message first goes west to column 0, then goes east column for column, where for each column one goes first completely to the north, then completely to the south. Therefore we take as measure a list $[w, e, n, s]$ such that each element denotes the maximum number of steps that can be taken in respectively the western, eastern, northern and southern direction. This list decreases under lexicographical ordering. Function $\boldsymbol{\mu}^{\mathtt{wf}}$ computes this measure. It takes as parameters the current port and the dimension of the network.

If we do a step in the western direction, the second element of the list increases. However, as the first element decreases, the measure as a whole decreases under lexicographical ordering. As long as we are heading in the western direction (denoted with $\leftarrow$), the other elements of the list actually do not matter and are set to zero.

Once we are heading in the eastern direction ($\rightarrow$), no more western steps can be done, because west-first routing routes in western direction first. Thus the first element w must be zero. In this case the second element e represents the number of steps that can be done in eastern direction. The remaining elements n and s do not matter as long as the packet is heading east. They are both set to zero.

Similarly, if we are heading in the northern direction ($\uparrow$), the number of steps that can be taken in the southern direction increases. However, the measure decreases as n – the number of steps that can be taken in the northern direction – decreases. Once we do a north-south turn, we are heading in the southern direction ($\downarrow$). No more steps can be taken in northern direction, thus n is set to zero. Element n remains zero until we do a step east in which case e decreases and n and s can be reset.

$$\boldsymbol{\mu}^{\mathtt{wf}}(c, dim) \stackrel{\text{def}}{=} \begin{array}{ll} [c.x, 0, 0, 0] & \text{if } \leftarrow \\ [0, dim.x - c.x, 0, 0] & \text{if } \rightarrow \\ [0, dim.x - c.x, c.y, dim.y] & \text{if } \uparrow \\ [0, dim.x - c.x, 0, dim.y - c.y] & \text{if } \downarrow \end{array}$$

Function $\boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}$ computes the switching measure. It is simply the pairwise addition of the measure for all enroute messages.

$$\boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}(\sigma, dim) \stackrel{\text{def}}{=} \sum_{t \in \sigma.E} \boldsymbol{\mu}^{\mathtt{wf}}(t.l, dim)$$

Switching measure: Proof

Proof obligation 11 uses some auxiliary theorems. First we prove that measure $\boldsymbol{\mu}^{\mathtt{wf}}$ indeed decreases under lexicographical ordering when a message advances from a current port to a next hop, i.e., we prove:

$$\boldsymbol{\mu}^{\mathtt{wf}}(n, dim) < \boldsymbol{\mu}^{\mathtt{wf}}(c, dim) \text{ for all } n \in \mathbf{R}^{\mathtt{wf}}(c, dest, dim)$$

This can be proven without further interaction. Secondly, we prove that measure $\boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}$ never increases after a call of the wormhole switching policy. For each travel t, assuming it remains enroute and is not consumed, the measure $\boldsymbol{\mu}^{\mathtt{wf}}$ is computed before and after switching. It never increases, intuitively meaning that messages are not switched further away from their destination.

$$\forall t \in \mathbf{S}^{\mathtt{whs}}(\sigma).E \cdot \boldsymbol{\mu}^{\mathtt{wf}}(\mathbf{S}^{\mathtt{whs}}(\sigma).E[t.id].l, dim) \leq \boldsymbol{\mu}^{\mathtt{wf}}(t.l, dim)$$

When the wormhole switching policy is called with a deadlock-free configuration, at least one message can move, i.e., one message can progress from its current location to a next hop. The first lemma states that for this message the switching measure decreases. For all other messages, the second lemma states that the measure does not increase. Combining the two auxiliary theorems, we prove Proof Obligation 11, i.e., that if the wormhole switching policy is called with a deadlock-free configuration, the switching measure decreases.

$$\neg\, \boldsymbol{\Omega}(\sigma) \wedge \sigma.E \neq [] \implies \boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}(\mathbf{S}^{\mathtt{whs}}(\sigma), dim) < \boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}(\sigma, dim)$$

Proof Obligation 12 states that the switching measure must be independent of the ordering of the enroute messages. This holds as function $\boldsymbol{\mu}^{\mathtt{wf}}_{\mathbf{S}}$ is a sum. Proof obligation 12 can easily be discharged.

Starvation measure: Definition

Let message t wait for next hop n. The characteristic of the FIFO starvation prevention mechanism is that each time n becomes available, message t advances in the list of all messages that wait for n. Therefore, we can take as measure the index of message t in this list. Let function $\text{index}(t, E)$ return the index of message t in the list of messages E. We define function $\boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}$ as follows:

$$\boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \sigma) \stackrel{\text{def}}{=} \text{index}(t, [t' \in \sigma.E \mid n \in \mathbf{R}(t'.l, t'.d)])$$

Function $\boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}$ filters all messages t' that wait for next hop n and returns the index of message t in this list.

Starvation measure: Proof

Proving correctness of the starvation measure requires some proof effort. We first prove that for any starving message t, i.e., any message t that has not moved in the call of switching method $\mathbf{S}^{\mathtt{whs}}$ even though a next hop n was available, the index of t in the list of messages waiting for n decreases.

$$\text{starving}(t) \implies \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \mathbf{S}^{\mathtt{whs}}(\sigma)) < \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \sigma)$$

We require an auxiliary lemma. The switching module tries to move each message in the list of enroute travels, starting with the first travel in the list. If message t did not move, a message t' preceding t in the list of enroute messages acquired the next hop. We prove that if a message preceding t is not starving, then the index of t in the list of enroute messages decreases.

$$\begin{aligned}\text{starving}(t) \wedge \neg\,\text{starving}(t') \wedge \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t', n, \sigma) < \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \sigma) \implies \\ \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \mathbf{S}^{\mathtt{whs}}(\sigma)) < \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \sigma)\end{aligned}$$

Then we prove that injecting new messages does not influence the measure of the enroute travels. If a message is injected, it is initially non-starving. Appending non-starving messages to the list does not influence the starvation measure of starving messages. Let function add_enroute append a message at the end of the list of enroute messages. We prove:

$$\text{starving}(t) \wedge \neg\,\text{starving}(t') \implies \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \text{add\_enroute}(t', \sigma)) = \boldsymbol{\mu}^{\mathtt{F}}_{\mathbf{O}}(t, n, \sigma)$$

This holds as function $\mathbf{O}^{\mathtt{F}}$ appends non-starving message t' at the end of the list, thereby not influencing the index of t. This concludes the proof of both Proof Obligations 15 and 16.

Deadlock measure

Proof Obligation 14 requires a proof that given a permanently blocked message, there exists a deadlock. Our definition of deadlock, i.e., predicate $\boldsymbol{\Omega}^{\mathtt{whs}}$, identifies deadlocks in which *all* messages are permanently blocked. We have to prove that if there exists a configuration in which at least one message is permanently blocked, there exists a configuration in which all messages are permanently blocked.

This proof has been done by Duato [48]. Assume a configuration σ in which the next hops of some message t are permanently unavailable. As there is no starvation, this is due to a local deadlock. We drain the configuration, by stopping all injections and by waiting for all messages not participating in the local deadlock to be consumed. This yields a deadlock configuration σ', in which all messages are permanently blocked, i.e., for which $\boldsymbol{\Omega}^{\mathtt{whs}}(\sigma')$ returns true.

Proof Obligation 14 assumes deadlock freedom. Consequently, the existence of deadlock configuration σ' is a contradiction. This means that the existence of configuration σ – in which the next hops of some message t are permanently unavailable – is a contradiction as well. Thus there does not exist a configuration σ in which the next hops of some message t are permanently unavailable. As for all configurations no message can be permanently blocked by unavailable next hops, there *exists* a deadlock measure that decreases with each step in which all next hops are unavailable. Proof Obligation 14 is discharged.

4.3.2 Deadlock Verification

Discharging Proof Obligation 7 is a major task. We reused an earlier proof effort on the theorem of Dally and Seitz that an acyclic dependency graph is sufficient for deadlock freedom [36]. In order to discharge Proof Obligation 7, we have defined the dependency graph $G^{\mathtt{wf}}_{\mathtt{dep}}$, proven that this dependency graph accurately reflects the waiting relations in the network, and proven the graph acyclic. The latter proof reused measure $\boldsymbol{\mu}^{\mathtt{wf}}$: as there is a strict partial ordering on the vertices of the dependency graph, the graph is acyclic.

First, we prove that if there exists a strict partial ordering $\sqsubset$ on the vertices of a graph, the graph is acyclic. We prove a lemma stating that for any path $\pi = v_0, v_1, \ldots v_k$ any vertex v_j $(0 < j \leq k)$ is smaller than the first vertex.

$$\mathrm{path}(\pi, G) \wedge 0 < j < |\pi| \implies v_j \sqsubset v_0$$

This lemma follows directly from the transitivity of the partial ordering. The proof proceeds by contradiction: assume there is some cycle $c = c_0, c_1, \ldots c_k$. We instantiate the lemma with $\pi = c_0, c_1, \ldots c_k, c_0$ and $j = k + 1$. Since c is a cycle, π is a path. The lemma proves that $c_0 \sqsubset c_0$. The contradiction follows from the irreflexivity of the strict partial ordering.

Next, we prove that if the vertices of the dependency graph are converted to lists of natural numbers using function $\boldsymbol{\mu}^{\mathtt{wf}}$, the lexicographical ordering is a strict partial ordering on the vertices of the dependency graph.

$$\boldsymbol{\mu}^{\mathtt{wf}}(n, dim) < \boldsymbol{\mu}^{\mathtt{wf}}(c, dim) \text{ for all } n \in A^{\mathtt{wf}}_{\mathtt{dep}}(c, dim)$$

This proof needs no further interaction.

Besides proving that the dependency graph is acyclic, we also need to prove correctness of the graph. We prove two theorems. First, we prove that if there is an edge (r_0, r_1) in the graph, there exists a destination d such that if a travel with destination d is located in r_0 it is routed to r_1.

$$r_1 \in A^{\mathtt{wf}}_{\mathtt{dep}}(r_0, dim) \implies \exists d \in P \cdot r_1 \in \mathbf{R}^{\mathtt{wf}}(r_0, d, dim)$$

For this proof we define a function which computes a witness destination d for the dependency. For west-first routing, this is simply the processing node closest to r_1. Secondly, we prove that if a travel can be routed from r_0 to r_1, there is a corresponding dependency edge. This proof is a straightforward case distinction.

Part of the proof for deadlock is general and could be re-used in other efforts. Proving that a graph is acyclic from a partial order is not specific to west-first routing, but should be part of some graph theory library. The same holds for the proofs of e.g., transitivity, trichotomy, and irreflexivity of the lexicographical ordering. These parts constitute approximately 168 lines of the total number of 834 lines required for this proof. However, the major part of the code is truly instantiation-specific. Discharging Proof Obligation 7, i.e., proving deadlock freedom, requires the most interaction of the user and constitutes the major part of the proof effort.

Chapter 5
Conclusion

In this chapter, we reflect on our methodology. First, we consider the correctness criterion presented in this part. Our definition of productivity takes finite executions into account only. Traditionally, starvation freedom is defined using infinite executions. We defend our definition and argue about the relevance of our correctness criterion by showing that productivity is commonly – but implicitly – used as an assumption in papers dealing with applications of communication networks. We proceed with a discussion on the strengths and weakness of the new GeNoC framework.

5.1 Definition of Productivity

Our definition of productivity is constructive and deals with an arbitrary but finite number of messages only. We prove theoretical worst-case finite bounds on the time it takes for messages to be consumed. If such a bound can be defined for each message m, any message m will eventually reach its destination, provided that this bound does not depend on the total number of uninjected messages. It will not be permanently blocked in a deadlock, it will not permanently move around in a livelock, or permanently wait due to a starvation scenario. Important is that these bounds are required to be independent of the communications that are still pending. Without this requirement, our finite definition would not correctly distinguish starvation-free networks from networks where starvation may occur.

Traditionally, starvation is defined using infinite executions [5]. In order for a starvation to occur, some infinite execution must occur where some process – in our case a message – never acquires the resources it requires, even though this resource becomes available infinitely often [48]. Even though the definition of starvation requires the notion of infinity, the definition of starvation freedom does not.

Figure 5.1 provides a network with a starvation scenario. The network has two sources src_1 and src_2 where messages are injected into queues. Source src_1 injects only one message m at the first clock cycle. Source src_2 eagerly injects a continuous stream of messages from the first clock cycle. Both queues are connected to an arbiter which merges the incoming messages in the queues. It grants the turn to one of its inputs and sends the message at the head of this queue to queue q. In

the next clock cycle, the message in queue q is sent to a sink where it is consumed immediately. The arbiter gives priority to the lower queue.

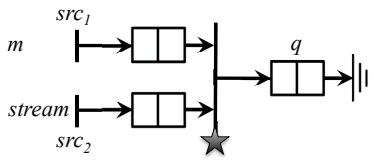


Figure 5.1: *Starvation Scenario*

As the arbiter gives priority to one of its inputs, it is not fair and does not prevent starvation. We describe the starvation that occurs. Queue q becomes available infinitely often. At the first clock cycle it is empty and thus available. At each odd clock cycle it is empty and able to receive a message from the stream injected by source src_2. At each successive even clock cycle it is full as it still has to transmit its message to the sink to be consumed. Message m is in a starvation scenario as it permanently waits to acquire queue q, even though this queue becomes available infinitely often. The network is not productive.

Even though we have formulated our definition of productivity for finite communications only, it correctly identifies this network as unproductive. Property 2 of our definition (see Page 31) – stating that any message spends a finite time in the network – does not hold. It is not possible to define a suitable bound on the time message m spends in the network, i.e., on how long it takes for m to be granted the turn by the arbiter.

Without the addition that the bound cannot depend on what messages are to be injected, the definition would identify the network as productive. As we consider finite communications only, the length of the stream at source src_2 is arbitrary but bounded by some number n. This means that at clock cycle $2n+1$ all messages from the stream have been injected, have moved through the network, and have been consumed. At this point, message m can move. Thus a bound can be defined on the time message m is in the network, namely $2n + 2$. This bound depends on n, i.e., it depends on the number of messages that is still to be injected. Our definition does not allow such bounds.

The term "productivity" has been borrowed from the stream community [131, 53, 52, 147]. In the stream community the term productivity captures an intuitive correctness property expressing that a stream where sufficient data is coming in is always able to produce correct data at the output. It represents the notion of "unlimited progression" [52]. More formally, a stream is productive if every nth element of the stream can be computed in finite time [147].

The notion of productivity in the stream community has similarities to our notion of productivity. They both state that always eventually some token or message will correctly and in finite time pass through some set of computations. A difference is that we define and prove productivity over finite data structures.

5.2 Productivity in Literature

We will now argue that our correctness criterion is a relevant and desired property of communication networks. The major part of the literature on applications built on top of communication networks assume the availability of a reliable network. A well-built network should appear as a wire to its clients [37, 15]. Processes that use the network to communicate assume that messages can be injected into the network, can traverse the network, and will arrive intact at the intended destination [96]. Such behavior is ensured by productivity.

Productivity as assumption in the design of application layer protocols

The major part of the literature on application layer protocols built on top of communication networks is on cache coherency. We provide some examples and provide details on the assumptions they make on the network.

Bolotin et al. propose a hardware-based mechanism for efficient distributed directory-based cache-coherent access for NoCs [16]. A round-trip of a request and response message between a processor and an L2-cache bank over the NoC may induce delay, but this delay is assumed to be finite. This assumption is tantamount to assuming productivity. They also assume that the network maintains the ordering of messages for each source-destination pair. Productivity does not deal with in-order packet delivery.

Pétrot et al. propose a software solution to the problem of sharing data in multiprocessor SoC's [114]. They assume a generic system architecture which enforces separation between IP functionality and IP communication. The interconnect is assumed to behave like it provides direct point-to-point communication channels between requesting nodes and target nodes. Assuming such bus-like behavior equates to assuming productivity.

Massas and Pétrot provide a comparison of different cache coherency protocols implemented on top of a NoC [39]. They characterize the protocol actions with a hop as atomic unit. A hop is defined as the delay needed to cross the NoC from one node to another. It is assumed that this delay is finite, i.e., that the interconnect can always inject a message and deliver it at its desired destination.

Marescaux et al. provide a comparison between different ways of achieving memory sharing on NoCs [92]. Their comparison is based on a specific NoC architecture with deterministic routing to guarantee deadlock freedom and in-order packet delivery.

Without exception, these works assume productivity to decouple the design and validation of application layer protocols from the underlying network architectures.

Productivity as assumption in the verification of application layer protocols

We provide examples on formal verification efforts related to application layer protocols. Again, we focus on cache coherency as this has been an active research field for many years. All methodologies mentioned in this section decouple the verification of the protocol from the verification of the interconnect. They all abstract

from the communication network by assuming availability of the interconnect and by assuming that messages are reliably delivered to their destination.

Model-checking has been successfully applied to the verification of high-level descriptions of cache coherency protocols. Symbolic model checking is applied to parameterized versions of write-invalidate and write-update cache coherency protocols [41, 42]. The verification is parametric in the number of processors in the system. Emerson and Kahlon apply a custom model checking method to verify snoopy cache coherence protocols such as MESI, MOESI, Illinois, Berkeley and Dragon [51]. Each protocol is verified within a fraction of a second. Similar results have been achieved with the Murphi model checker [30, 108, 26]. All these studies verify safety properties at a high level of abstraction.

McMillan verifies both safety and liveness properties [95]. He uses the SMV model checker to support a counterexample driven methodology. The approach first naively tries to prove a property and when a counterexample arises, it diagnoses the cause of the error and rules it out. SMV has also been used to verify both safety and liveness properties [50]. Baukus et al. model a parameterized system as a higher order transition system in a decidable logic [7]. These transition systems are given to a model checker to check both safety and liveness properties.

Another verification approach is theorem proving. Moore formalized a simple write-invalidate cache scheme using the ACL2 theorem proving system [79]. An arbitrary number of processor-cache pairs interacts with a global memory via a bus which is snooped by the caches. Moore models both the cache system and the memory system without cache as first order functions and proves that executing read and writes in the system with caches equals executing in the system without caches. Park and Dill apply the PVS theorem prover to verify a directory-based cache coherence protocol developed for the Stanford FLASH multiprocessor [113]. Their method compares a state graph representing the implementation of the protocol with a state graph representing the desired abstract behavior. Some hybrid methods have been proposed, combining theorem proving with model checking [120].

SMT solvers are applied in [111]. They generate invariants to verify directory based protocols using a solver for first order logic with uninterpreted functions (EUF).

The efforts reported so far focus on applying general verification techniques to cache coherency protocols. Application-specific tools have been proposed as well. Plakal et al. present a special reasoning technique that assigns timestamps to relevant protocol events [117]. These timestamps are used to construct a total ordering of events. These total orderings are used to verify that the requirements of a particular memory consistency model have been satisfied. Pong and Dubois propose the tool SSM [118]. It is a state-based verification tool based on an abstraction technique preserving the properties to verify. Recently, Zhang et al. proposed a correct by construction approach [148].

With exception of two recent papers [108, 26], processes communicate through a bus instead of a network. However, O'Leary et al. abstract this network into a set of point-to-point channels.

Productivity is commonly used as assumption in the design and verification of application layer protocols. Assuming productivity is used commonly to decouple the application layer from the network layer.

5.3 The GeNoC Framework

We address the applicability, the restrictions and the usability of our methodology.

Applicability of GeNoC

The generality of our approach ensures a wide applicability. All proof obligations have been defined to be as weak as possible. They apply to a large family of communication networks. Section 2.3.2 (see Page 26) gives some examples of different instantiations from the literature.

Switching can be instantiated with different flavors of packet, wormhole and circuit switching. Injection can be, e.g., greedy, based on a per message desired injection time, or credit-based. The starvation prevention mechanism can be instantiated with, e.g., FIFO or round-robin arbitration. Routing and topology can be instantiated with many different functions.

As Table 3.6 shows (see Page 50), livelock freedom depends on both the routing function and the switching policy. Some networks are livelock-free because of the routing function, e.g., the routing function prevents messages from going into a cycle. In such networks, the switching measure could be the sum of the lengths of the routes of all messages. Some networks are livelock-free because of the switching policy, e.g., networks where the time a message can spend in the network depends on a counter in the header of the message [132]. In such networks, the switching measure could be the sum of the counters. Both measures are sums and thus irrelevant of the order of the messages. For both measures, Proof Obligations 11 and 12 can be discharged and thus both types of networks can be proven livelock-free with our methodology.

Restrictions of GeNoC

The proof obligations enforce some limitations. First, the consumption assumption is required. A message arriving at its destination is eventually consumed. Without the consumption assumption, message dependencies occur [68] which can currently not be modelled in GeNoC. The current definition of functional correctness (Definition 3.4 on Page 40) does not deal with dropping, duplication or joining of packets. This prevents modelling of synchronizations in the network. Each message is assumed to have one destination, making verification of multicast and broadcast algorithms difficult. Finally, the network model is assumed to be static. Dynamic failure of, e.g., channels or switching in the network cannot be represented.

Usability of GeNoC

We have shown in Chapter 4 that for a non-trivial example, approximately 86% of the proof of productivity can be derived automatically from the GeNoC framework. Part of this is due to the fact that those proof obligations that depend only on the switching policy have been discharged once and for all for both packet- and wormhole switching. These are the most commonly used switching policies. For any network with either packet- or wormhole-switching, these proofs can be reused without any modification. Our ultimate goal is to develop a library containing most common injection methods, switching techniques, and starvation prevention mechanisms. These components are easily reusable.

Proof Obligation 7, which states that the network must be deadlock-free, is hardest to discharge. The remainder of this thesis will focus on this proof obligation. We have examined weak sets of proof obligations under different network models. The weakest possible proof obligation is a necessary and sufficient condition. Chapters 6 presents necessary and sufficient conditions for deadlock freedom. These conditions can replace Proof Obligation 7.

Still, discharging these necessary and sufficient conditions for some specific network is a difficult task. It requires a great deal of manual theorem proving. These proofs are not easily reused for other instantiations. Chapters 7 and 9 present algorithms that automatically prove deadlock freedom. For fixed-size networks, one can simply use these algorithms to discharge Proof Obligation 7.

In Chapter 8, the tool DCI2 is presented, which checks deadlock freedom and discharges all proof obligations related to the topology, the resources and the routing function. DCI2 allows a user to automatically discharge all proof obligations – and thereby prove productivity – for packet- and wormhole-switched networks. In combination with an ACL2 library of common network modules, DCI2 provides a completely automatic way of proving productivity for a large family of communication networks.

We presented a formal specification and verification environment for high-level descriptions of communication networks. The environment supports the proof of safety and liveness properties. Proofs are performed for parametric descriptions, i.e., the number of nodes and the size and the number of messages are all left uninterpreted. Key is the reusability of proofs.

Productivity has been broken down into five network properties: functional correctness, deadlock freedom, livelock freedom, starvation freedom and liveness of injection. Except for deadlock freedom, each of these properties has been broken down into simple and elementary proof obligations. Breaking down deadlock freedom into smaller proof obligations is a hard problem. The next part of this thesis is devoted to automatic ways of proving deadlock freedom of communication networks.

Part III

Isolated Network Layer Deadlock Verification

Chapter 6
Necessary and Sufficient Conditions for Deadlock-free Routing

This chapter discusses new necessary and sufficient conditions for deadlock-free routing in packet and wormhole networks. A detailed analysis of existing theorems will reveal the relevance of our new conditions. Existing conditions are often counterintuitive and require many different concepts and definitions. We will expose a subtle discrepancy in Duato's seminal condition for deadlock-free routing in wormhole networks [44], showing that it is not complete. In contrast, the conditions presented in this chapter are simple, correct and require only a few concepts. This makes it possible to discharge them automatically using dedicated algorithms. Our conditions satisfy the following properties:

Static A static condition does not consider the dynamic evolution of the network. It does not consider the state of the network at a given time. Dally and Seitz' condition is static, as it solely depends on the dependency graph [36]. This graph can statically be computed from a specification of the network topology and the routing function. The major benefit of a static condition is that it is easier to discharge. Even for a small network, analysis of all possible states, i.e., all possible injection sequences and all possible configurations that can be reached during the evolution of such an injection sequence can be impossible. In contrast, Dally and Seitz' static condition can be discharged just by computing the dependency graph and searching for cycles.

Wide Applicability Our conditions have as few assumptions as possible, widening their applicability. We do not require routing to be coherent (see Section 6.7.1 for a discussion on coherency). Defining a static necessary and sufficient condition for incoherent routing functions has been an open problem up to now [48]. We do not require the routing function to prevent livelocks. Livelocks can be prevented by other mechanisms such as hop counters or packet dropping. Our conditions separate the proof of deadlock freedom from the proof of livelock freedom and enable the verification of networks employing such mechanisms.

Mechanically proven We have mechanically proven necessity and sufficiency of our conditions for absence of deadlocks using the ACL2 theorem prover. This enabled us to get all definitions and proofs completely correct.

The first part of this chapter deals with packet networks. The condition presented in this thesis is based on the notion of *escapes*. It states that "in any set of channels there is at least one escape" is necessary and sufficient for the absence of deadlocks. Of course, this depends on the definition of "escape". We formalize the exact requirements for a channel to be an escape. A discussion of our definition of deadlock will show that it is irrelevant whether channels are implemented with queues or buffers.

The second part of this chapter deals with wormhole networks. Again, our condition states that in any set of channels there must be at least one escape. However, a channel that is an escape in a packet configuration is not necessarily an escape in a wormhole network. We will stress the differences between the conditions. Again, we discuss our definition of deadlock. This discussion will show that our simplified definition of deadlock is logically equivalent to the more complicated one used in the literature.

6.1 Notation and Definitions

All chapters in this part share the notations and definitions presented in this section. At the end of this section, an overview is given.

We start by giving the formal definition of a network. This definition refines the network model presented in Section 3.2.2 (see Page 35).

Definition 6.1 A *network* N is a tuple with a finite set of processing nodes P, a finite set of arcs C, a routing function $\mathbf{R} : P \times P \mapsto \mathcal{P}(C)$, a domain of flits F, a domain of message identifiers M, two topology functions $\mathrm{src} : C \mapsto P$ and $\mathrm{end} : C \mapsto P$, a capacity function $\mathrm{cap} : C \mapsto \mathbb{N}$, a function $\mathrm{msg} : F \mapsto M$, and a function $\mathrm{dest} : M \mapsto P$.

$$N ::= \langle P, C, \mathbf{R}, F, M, \mathrm{src}, \mathrm{end}, \mathrm{cap}, \mathrm{msg}, \mathrm{dest} \rangle$$

The topology of network N is defined by functions $\mathrm{src} : C \mapsto P$ and $\mathrm{end} : C \mapsto P$. These functions return the processing nodes at the source and at the end of a channel, respectively. The routing function computes from each processing node s and each destination node d a set of next hops $\mathbf{R}(s, d)$. This set is non-empty if $s \neq d$. Each channel c has a certain number of places, where each place can either be empty or store exactly one flit. An empty place is denoted with ϵ. We assume $\epsilon \notin F$ and let $F_\epsilon \stackrel{\mathrm{def}}{=} F \cup \{\epsilon\}$ denote the domain of flits plus the empty place. The capacity of channel c, denoted $\mathrm{cap}(c)$, is defined as the number of places of channel c.

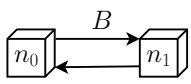
Flits are the atomic units of transfer that are transmitted through channels, i.e., they are concrete pieces of data that are to be communicated. At any time, channels may contain many non-unique flits. Each flit belongs to at most one message, but each message may consist of multiple flits. In our definitions and proofs, we need to be able to distinguish equal flits belonging to different messages.

To this end, the set of unique message identifiers M is used. Function msg is able to retrieve, given a flit f, the unique message id of the message to which flit f belongs. Since flits are not unique, some additional bookkeeping is required for the actual implementation of this function. Finally, function dest returns the destination of the message corresponding to the given message identifier.

For sake of presentation, we introduce some shorthand notations. Let f be a flit, and let list L be a subset of F_ϵ.

$$\begin{aligned} \text{dest}(f) &= \text{dest}(\text{msg}(f)) \\ \text{dests}(L) &= \{d \in P \mid \exists f \in L - \epsilon \cdot \text{dest}(f) = d\} \\ \text{msgs}(L) &= \{m \in M \mid \exists f \in L - \epsilon \cdot \text{msg}(f) = m\} \end{aligned}$$

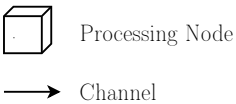
Figure 6.1a gives an example of a network. There are two processing nodes $P = \{n_0, n_1\}$, and two channels $C = \{A, B\}$. Figure 6.1b shows the routing function $\mathbf{R}$. The set of flits F consists of tuples $\langle data, header \rangle$ representing a packet with data that is to be transmitted and a header that contains the destination of the packet. Each packet can be uniquely identified by some natural number, i.e., $M = \mathbb{N}$. The topology functions src and end can be derived from Figure 6.1a. The capacity function cap assigns 3 places to each channel. Function dest looks up the packet referred to by id n and returns its header.



(a) *Network*

n	$\mathbf{R}(n, n_0)$	$\mathbf{R}(n, n_1)$
n_0		B
n_1	A	

(b) *Routing Function*



(c) *Legend*

Figure 6.1: *Example of network*

Since channels store messages and there are no further state holding components in the network, the state of the network is determined completely by the state of the channels. A state or configuration contains information on which flits are in which places.

Definition 6.2 A *configuration* σ is an assignment of lists of flits or empty places to channels.

$$\sigma : C \mapsto \mathcal{L}(F_\epsilon)$$

Function σ returns a *list* of flits. The list of flits in channel c can be accessed through $\sigma(c)$. This list contains the flit stored in the head of the channel first, followed by flits stored in the tail of the channel. That is, given a channel c, the flit which occupies the head of the channel is returned by $\sigma(c)[0]$. The number of flits in c is denoted $|\sigma(c)|$. The empty configuration is denoted by σ_ϵ. The set of all configurations is denoted by Σ.

Given a configuration σ and a message id m, it is possible to compute the set of channels that is occupied by the message corresponding to message id m. This set of channels is denoted by channels(m, σ).

Not any assignment of flits to channels constitutes a legal configuration. Consider the network in Figure 6.1a. A possible configuration is:

$$\sigma(A) = [f_1, f_1, f_1] \quad \text{where} \quad \text{dest}(f_1) = n_1$$
$$\sigma(B) = [f_0, f_0, f_0] \quad \text{where} \quad \text{dest}(f_0) = n_0$$

This configuration assigns flits belonging to a message destined for processing node n_1 to channel A. However, as processing nodes consume messages as soon as they arrive, channel A can never contain such flits. We introduce function $\tau : C \mapsto \mathcal{P}(P)$ which given a channel returns the typing information of that channel, i.e., which types of flits can be in the channel. In other words, if d is a type of channel c, channel c is *reachable* by messages destined for d. As example, the typing information of channel A in Figure 6.1a is the set $\{n_0\}$, as only flits belonging to messages destined for processing node n_0 can ever reach channel A. A flit with destination d can reach channel c if and only if the processing node at the source of c routes flits destined for d towards c.

Definition 6.3 The *typing information of channel* c, notation $\tau(c)$ is defined as the set of destinations for which there exists a message that can be in channel c.

$$\tau(c) \quad \stackrel{\text{def}}{=} \quad \{d \in P \mid c \in \mathbf{R}(\text{src}(c), d)\}$$

When drawing a network, we will sometimes add the typing information of the channels. In this case, each channel name will be followed by a semicolon and the typing information. For example, the network in Figure 6.1a will additionally have the following information:

$$A \ : \ n_0$$
$$B \ : \ n_1$$

A routing path is a simple path established by the routing function for messages destined for destination d.

Definition 6.4 Let $d \in P$ be a destination in the network. Path π^d is a *routing path for destination* d, notation $\mathbf{R}$-path(π^d), if and only if π^d is a list of channels $[c_0, c_1, \ldots, c_k]$ that constitute a route for a message destined for d.

$$\mathbf{R}\text{-path}(\pi^d) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{ll} & c_0 \in \mathbf{R}(\text{src}(c_0), d) \\ \wedge & \forall 0 < i \leq k \cdot c_i \in \mathbf{R}(\text{end}(c_{i-1}), d) \\ \wedge & \forall 0 \leq i, j \leq k \cdot i \neq j \implies c_i \neq c_j \end{array} \right.$$

We denote a set of routing paths $\{\pi^{d_0}, \pi^{d_1} \ldots, \pi^{d_k}\}$ with $\Pi^\star$.

A dependency between two channels c_0 and c_1 indicates that in some configuration a packet in c_0 waits for channel c_1 to become available. The dependency graph reflects all dependencies in the network.

Definition 6.5 The *dependency graph* $G_{\texttt{dep}}$ of network N is a directed graph with as vertices the set of channels C. Function $A_{\texttt{dep}} : C \mapsto \mathcal{P}(C)$ represents the arcs of the dependency graph. There is an arc between vertex c_0 and c_1 if and only if there is a routing dependency between these channels:

$$A_{\texttt{dep}}(c_0) \quad \stackrel{\text{def}}{=} \quad \{c_1 \mid \exists d \in \tau(c_0) \cdot c_1 \in \mathbf{R}(\text{end}(c_0), d)\}$$

For example, the dependency graph of the network in Figure 6.1a consists of two vertices, namely channel A and B. There are no edges, indicating that there are no waiting relations between the channels.

A dependency (c_0, c_1) is *caused by* destination d if and only if d is in the typing information of c_0 and if $c_1 \in \mathbf{R}(\text{end}(c_0), d)$. When drawing a dependency graph, the destinations causing the dependencies will be used to label edges. We will overload function $A_{\mathtt{dep}}$ so that if an extra parameter d is given, it returns only the dependencies caused by destination d. Finally, function $\boldsymbol{\tau}(c_0, c_1)$ returns the labels of dependency edge (c_0, c_1), i.e., the set of destinations causing the dependency.

Table 6.1 provides an overview of the definitions and notation used in this part.

Description	Notation
Processing nodes	P
Channels	C
Source of channel	$\text{src}(c)$
End of channel	$\text{end}(c)$
Routing	$\mathbf{R}(s, d)$
Channel capacity	$\text{cap}(c)$
Empty place	ϵ
Domain of flits	F
Domain of flits plus empty place	F_ϵ
Domain of message identifiers	M
Message id of flit	$\text{msg}(f)$
Destination of message	$\text{dest}(m)$
Destination of message of flit	$\text{dest}(f)$
Destinations of list of flits	$\text{dests}(F)$
Messages of list of flits	$\text{msgs}(F)$
Configuration	σ
Empty configuration	σ_ϵ
Domain of configurations	Σ
Channels occupied by message	$\text{channels}(m, \sigma)$
Routing path	$\mathbf{R}\text{-path}(\pi^d)$
Set of routing paths	$\Pi^\star$
Typing information	$\tau(c)$
Channel c has type T	$c : T$
Dependency graph	$G_{\mathtt{dep}} = \langle C, A_{\mathtt{dep}} \rangle$
Destinations causing a dependency	$\boldsymbol{\tau}(c_0, c_1)$

Table 6.1: *Overview of definitions and notation*

6.2 Packet Switching: Formal Condition

A deadlock configuration is a legal configuration that is reachable from the initial empty configuration. The behavior of packet switching induces some constraints on which configurations are legal and which are not. As each channel with capacity n can store at most n packets and as packets are the atomic units of transfer, a legal configuration σ cannot assign more than $\mathrm{cap}(c)$ packets to channel c. Secondly, if a packet p is assigned to channel c, channel c must be reachable by packets with destination $\mathrm{dest}(p)$.

Definition 6.6 In a packet network a configuration σ is *legal*, notation $\mathrm{legal}^{\mathfrak{ps}}(\sigma)$, if and only if for all channels the capacity is not exceeded and all packets can be routed towards their current channel.

$$\mathrm{legal}^{\mathfrak{ps}}(\sigma) \stackrel{\mathrm{def}}{=} \forall c \in C \cdot |\sigma(c)| = \mathrm{cap}(c) \wedge \mathrm{dests}(\sigma(c)) \subseteq \tau(c)$$

If it is clear from the context that the definition concerns packet networks, the superscript $\mathfrak{ps}$ will be omitted.

Definition 6.7 For packet network N, a configuration σ is a *deadlock configuration*, notation $\mathbf{\Omega}^{\mathfrak{ps}}(\sigma)$, if and only if it is a legal and non-empty configuration where all packets are blocked.

$$\begin{aligned}\mathbf{\Omega}^{\mathfrak{ps}}(\sigma) \stackrel{\mathrm{def}}{=}\ & \mathrm{legal}(\sigma) \wedge \\ & \sigma \neq \sigma_\epsilon \wedge \\ & \forall c \in C \cdot \forall f \in \sigma(c) - \epsilon \cdot \forall n \in \mathbf{R}(\mathrm{end}(c), \mathrm{dest}(f)) \cdot |\sigma(n) - \epsilon| = \mathrm{cap}(n)\end{aligned}$$

For any packet f in any channel c, all next hops n supplied by the routing function from the processing node connected to the end of c to the destination of f must be full.

Even though not explicitly stated, Definition 6.7 correctly deals with the following aspects:

- Any deadlock is actually reachable;
- A configuration in which some part of the network is in deadlock, but in which some other set of messages can still move is correctly recognized as a deadlock;
- The definition is correct both for packet networks where channels are implemented with buffers and with queues.

Section 6.3 provides lemma's which justify Definition 6.7 and which allow us to proceed with this trimmed definition.

6.2.1 Our Condition

In a deadlock configuration, the set of channels that are involved in the deadlock forms a *subgraph* of the dependency graph. We consider subgraphs as a subset of the vertices, removing edges only if the source or end of the channel is not in the subset. An escape for a subgraph is a channel where any packet can be routed

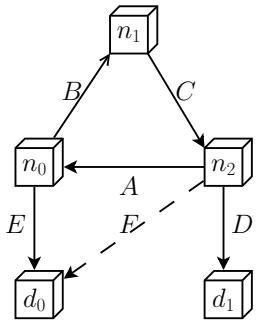
outside of the subgraph, regardless of the destination. This is either because messages are routed to channels not in the subgraph, or because messages arrive at their destination to be consumed. A channel with no dependency neighbors at all is a channel where any message arrives at its destination. We will call such channels *sinks*.

Definition 6.8 Given a subgraph S, a channel $e \in S$ is an *escape for* S, notation $\text{esc}(e, S)$ if and only if it is a sink or if for all possible reachable destinations there exists a dependency neighbor that is not contained in S:

$$\text{esc}(e, S) \quad \stackrel{\text{def}}{=} \quad e \in S \wedge (A_{\texttt{dep}}(e) = \emptyset \vee \forall d \in \tau(e) \cdot A_{\texttt{dep}}(e, d) \nsubseteq S)$$

Using two examples, we will show that a deadlock can never contain an escape and conversely, that if some subgraph has an escape, this set of channels does not form a deadlock. The first example has a deadlock, the second is deadlock-free.

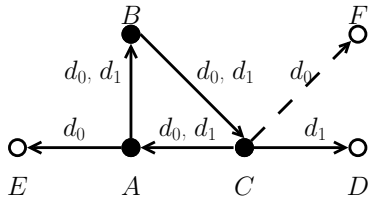
Example 6.1 Consider the interconnection network in Figure 6.2a without the dashed channel F. The network consists of five processing nodes. For sake of clarity, only processing nodes d_0 and d_1 are destination nodes. The routing function is depicted in Figure 6.2b. There is exactly one deadlock possible, in subgraph $S = \{A, B, C\}$. If channel A is filled with messages destined for d_1, channel B is full, and channel C is filled with messages destined for d_0, a circular wait occurs. This circular wait is represented by the cycle in the dependency graph in Figure 6.2c. None of the messages can be routed outside of the cycle. As subgraph S has no escape, there is a deadlock.



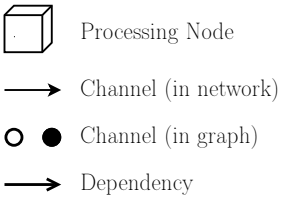
(a) *Interconnection network*

n	$\mathbf{R}(n, d_0)$	$\mathbf{R}(n, d_1)$
n_0	B, E	B
n_1	C	C
n_2	A	A, D

(b) *Routing Function*



(c) *Dependency graph*



(d) *Legend*

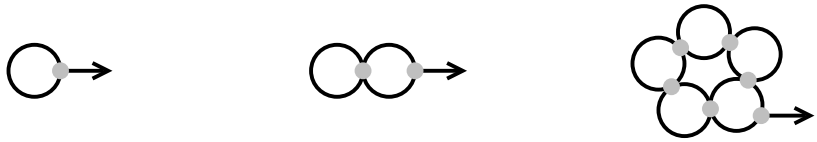
Figure 6.2: *Example of deadlock in a packet network. The dashed arrows are used in Example 6.2 only.*

Example 6.2 Consider again the interconnection network in Figure 6.2a, including the dashed channel F. Routing is extended with $\mathbf{R}(n_2, d_0) = \{A, F\}$. The network is deadlock-free. Subgraph $S = \{A, B, C\}$ can contain packets participating in a circular wait. However, channel C is an escape for subgraph S. For all destinations, i.e., for both d_0 and d_1, there is a dependency neighbor not in subgraph S. Any message in C can be routed outside of the subgraph. As subgraph S has an escape, it cannot form a deadlock.

If a channel is in deadlock it must either be in some dependency cycle or there must exist a path that leads to some deadlocked cycle. The condition is based on the idea that as long as dependency cycles have an escape, there is always a way to prevent the creation of deadlocks. Assume a cycle of full channels, i.e., a circular wait. As long as such a cycle has an escape there is always at least one message with a next hop outside the cycle. If all sets of such cycles have an escape, it is always possible to prevent deadlocks. Hence, the following condition:

A packet network is deadlock-free if and only if all sets of cycles in the dependency graph have an escape.

Figures 6.3a, 6.3b, and 6.3c illustrate the condition. Assume a deadlock-free network such that its routing function has a cyclic dependency graph. Figure 6.3c shows the strength of the theorem. At some point, the escape of a cycle may lead to next hops which are included in cycles which have already been escaped. The theorem states that this set of cycles again has an escape, so that at least one message will eventually be able to escape this cycle of cycles.



(a) *The escape leads messages out of the cycle.*

(b) *The escape leads to a cycle with a new escape.*

(c) *The escape leads to a cycle of cycles with a new escape.*

Figure 6.3: *Sets of cycles with escapes*

The necessary and sufficient condition is formulated as follows:

Theorem 6.1 A packet network is deadlock-free if and only if all subgraphs have an escape.

$$\mathsf{DLF}(N) \iff \forall S \subseteq C \cdot S \neq \emptyset \implies \exists e \in S \cdot \mathrm{esc}(e, S)$$

Theorem 6.1 is logically equivalent to stating that all sets of cycles have an escape. This formulation is easier to formalize and to prove.

6.2.2 Proof

Our proof transforms configurations to graphs in such a way that a deadlock always yields a graph with a knot[1]. A knot is a subgraph where every vertex has at least

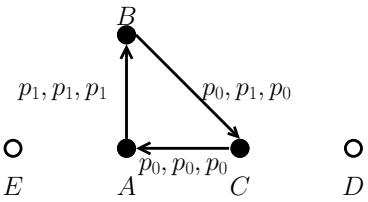
[1]Knots have proven to be a useful concept in deadlock detection [97].

one outgoing edge, and all outgoing edges from vertices in the knot end in the knot itself (see Appendix A.4 for a formal definition). Consider the following deadlock configuration σ for the network in Figure 6.2a:

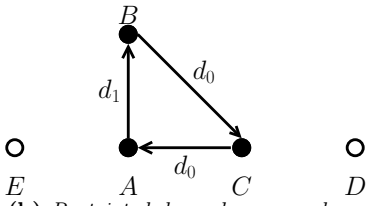
$$\begin{aligned}\sigma(A) &= [p_1, p_1, p_1] \\ \sigma(B) &= [p_0, p_1, p_0] \\ \sigma(C) &= [p_0, p_0, p_0]\end{aligned}$$

where p_0 and p_1 denote packets destined for d_0 and d_1 respectively

Figure 6.4a shows the first step of this transformation. Given a configuration, we define the *waiting graph*. This graph depicts the waiting relations in configuration σ. There is a waiting edge if and only if a packet is waiting to acquire a channel. A deadlock always yields a knot in the waiting graph: as all packets wait for each other, all waiting edges point to each other.



(a) *Waiting graph. The labels denote the packets that are waiting.*



(b) *Restricted dependency graph.*

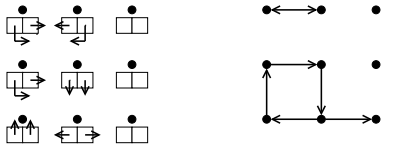
Figure 6.4: *Transformation from deadlock to knot in a graph. Per channel only one destination is considered.*

The waiting graph is not static as it depends on which packets are assigned to which channels in configuration σ. It is only used as an intermediary between a configuration and a knot in a static graph. Figure 6.4b shows the second step of the transformation. We show that if the dependency graph is restricted to contain dependencies caused by one destination per channel only, this *restricted dependency graph* is a subgraph of the waiting graph. Thus, given a deadlock, this graph contains a knot as well.

This completes the proof, as a knot in the restricted dependency graph is a subgraph that has no escape. Since for each channel in the knot there is one destination that leads back into the knot, there is no channel in the knot for which all destinations lead out of the knot. The knot has no escape. We have therefore proven the contrapositive version of Theorem 6.1: given a deadlock, there exists a subgraph without an escape.

Definitions

The waiting graph is dynamically defined by configuration σ. Informally, two channels c_0 and c_1 are connected in the waiting graph if there is packet in a place of c_0 that is routed to c_1. Figure 6.5 gives an example.

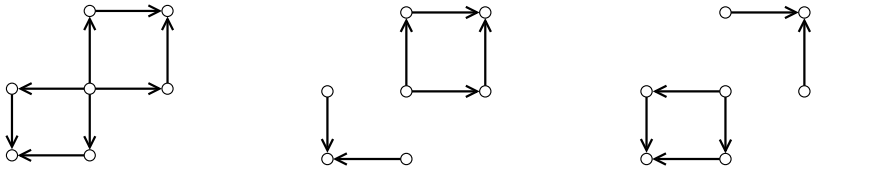


(a) *Configuration* **(b)** *Waiting graph*

Figure 6.5: *An example configuration and its waiting graph. In the configuration each channel has two places. Each arrow points to the next hop of the packet in the place.*

Definition 6.9 Given a configuration σ, the *waiting graph* is defined by set of vertices C and arc function $A^{\sigma}_{\texttt{wait}}$. There is an arc $(c_0, c_1) \in A^{\sigma}_{\texttt{wait}}$ if and only if there exists a packet in a place of c_0 with c_1 as next hop.

In the proof we consider subgraphs of the dependency graph defined by some restriction function $\delta : C \mapsto P$. This restriction function maps channels to destinations. At each channel c, it restricts the dependency graph to arcs leading to destination $\delta(c)$. Figure 6.6 gives an example.



(a) *Dependency graph* **(b)** *Restricted dependency graph* **(c)** *Restricted dependency graph*

Figure 6.6: *Let B and C be the only destinations. There are two δ-restricted dependency graphs: graph 6.6b is defined by $\delta(A) = B$; graph 6.6c is defined by $\delta(A) = C$.*

Definition 6.10 Given a restriction function $\delta : C \mapsto P$, the *δ-restricted dependency graph* is defined by set of vertices C and arc function $A^{\delta}_{\texttt{dep}}$. There is an arc $(c_0, c_1) \in A^{\delta}_{\texttt{dep}}$ if and only if $c_1 \in A_{\texttt{dep}}(c_0, \delta(c_0))$.

The Proof

We prove that there is a deadlock if and only if there exists a δ-restricted dependency graph with a knot. Assume a deadlock configuration σ. We show that this deadlock configuration implies a knot S in the waiting graph of σ (Lemma 6.1). We then construct a restriction δ, such that S is a knot in the δ-restricted dependency graph as well (Lemma 6.2). Assume a restriction δ and a knot S. We can construct a deadlock configuration by filling all channels in S with packets destined for the destinations provided by δ (Lemma 6.3). Hence, the contrapositive version of our theorem has been proven.

Lemma 6.1 A deadlock configuration σ implies there exists a knot S in the

waiting graph of σ.

$$\forall \sigma \in \Sigma \cdot \mathbf{\Omega}(\sigma) \quad \Longrightarrow \quad \exists S \subseteq C \cdot \text{knot}(S, G^{\sigma}_{\texttt{wait}})$$

Proof. Take as S the set of full channels in σ. We prove that S is a knot by contradiction. All the waiting-graph neighbors of S are full, since otherwise there would exist a channel with a packet with a next hop that is not full. This would imply that this packet can move, which contradicts the assumption of deadlock. Since all waiting-graph neighbors of S are full, and since S is the set of all full channels, S is a knot. □

The deadlock does not necessarily consist of full channels only: there can be a packet in an available channel, as long as each next hop for this packet is full. However, these available channels are not needed for the deadlock: the deadlock can be reduced to a set of full channels only, while preserving the fact that each packet is stuck.

Lemma 6.2 A knot S in the waiting graph of σ implies there exists a restriction δ such that S is a knot in the δ-restricted dependency graph.

$$\forall S \subseteq C \cdot (\exists \sigma \in \Sigma \cdot \text{knot}(S, G^{\sigma}_{\texttt{wait}}) \quad \Longrightarrow \quad \exists \delta : C \mapsto P \cdot \text{knot}(S, G^{\delta}_{\texttt{dep}}))$$

Proof. Choose δ such that for all channels $c \in S$, $\delta(c)$ returns the destination of one of the packets that is located in c in configuration σ. The set of neighbors created by this destination is a subset of the waiting-graph neighbors of c – and thus also a subset of S – since the waiting-graph neighbors include all next hops created by the destinations of all packets in c. □

Lemma 6.3 A knot in the δ-restricted dependency graph implies the existence of a deadlock configuration.

$$\exists S \subseteq C \exists \delta : C \mapsto P \cdot \text{knot}(S, G^{\delta}_{\texttt{dep}}) \quad \Longrightarrow \quad \exists \sigma \in \Sigma \cdot \mathbf{\Omega}(\sigma)$$

Proof. Construct a configuration σ by filling each channel $c \in S$ completely with packets destined for $\delta(c)$. The set of next hops of a packet in c is the set of dependency neighbors created by destination $\delta(c)$. Since this is a subset of S and since each channel in S is filled completely, all next hops of all packets are unavailable. Thus the configuration is in deadlock. □

From Lemma's 6.1 to 6.3 it follows that there exists a deadlock if and only if there exists a restriction δ such that the corresponding restricted graph has a knot. The existence of an escape for all subgraphs is exactly the negation of the existence of such a knot.

$$\begin{aligned}
&\exists S \subseteq C \cdot \exists \delta : C \mapsto P \cdot \text{knot}(S, G^{\delta}_{\texttt{dep}}) \\
&\quad = \quad \exists S \subseteq C \cdot \exists \delta : C \mapsto P \cdot S \neq \emptyset \wedge \forall c \in S \cdot A^{\delta}_{\texttt{dep}}(c) \neq \emptyset \wedge A^{\delta}_{\texttt{dep}}(c) \subseteq S \\
&\quad \Longleftrightarrow \quad \neg \forall S \subseteq C \cdot \forall d \in P \cdot S \neq \emptyset \implies \exists c \in S \cdot A_{\texttt{dep}}(c) = \emptyset \vee A_{\texttt{dep}}(c, d) \nsubseteq S \\
&\quad = \quad \neg \forall S \subseteq C \cdot S \neq \emptyset \implies \exists e \in S \cdot \text{esc}(e, S)
\end{aligned}$$

Thus a new necessary and sufficient condition for deadlock-free routing has been obtained: all subgraphs must contain an escape.

6.3 Definition of Deadlock

We justify our definition of deadlock. We prove Lemma 6.4 to show that it is not necessary to include reachability in Definition 6.7. Static analysis suffices to check for the absence of deadlocks, i.e., expensive reachability analysis is not necessary. Lemma 6.5 is used to prove that it suffices to check for deadlocks in which *all* packets are permanently blocked. Finally, Lemma 6.6 shows that Definition 6.7 applies both to networks with buffers and networks with queues.

Reachability

Duato has informally proven a lemma stating that any legal configuration is reachable [45]. This lemma allows us to omit reachability from Definition 6.7. Here, we formalize the lemma and its proof. This requires a formalization of reachability, and thus a transition relation. Such a transition relation contains two types of transitions. First, transitions that take a configuration and move packets from channel to channel according to the routing function and the semantics of packet switching. Secondly, transitions that take a configuration and inject a packet. Formalizing the first type of transitions is quite involved. It depends among others on whether packets move synchronously or asynchronously, and on the implementation of the channels. For the proof of the reachability lemma, it is only necessary to formalize the second type of transitions. Since routing is memoryless, we can consider each packet as injected by the source node of its current channel [45]. This formalization makes use of function $\text{repl}(L, l_1, l_2)$ which takes a list L and replaces the rightmost occurrence of l_1 with l_2, if $l_1 \in L$.

Definition 6.11 The transition relation $\xrightarrow{\mathfrak{ps}} \subseteq \Sigma \times \Sigma$ is defined by the following transitions:

$$\sigma \xrightarrow{\mathfrak{ps}} \sigma' \quad \text{if} \quad \sigma' \text{ is the result of moving one or more injected packets from their current channel to a next hop}$$

$$\sigma \xrightarrow{\mathfrak{ps}} \sigma' \quad \text{if} \quad \exists c_i \in C \cdot \begin{cases} \forall c \in C \cdot c \neq c_i \implies \sigma'(c) = \sigma(c) \\ \sigma'(c_i) = \text{repl}(\sigma(c_i), \epsilon, f) \text{ with } \mathbf{R}(\text{src}(c_i), \text{dest}(f)) = c_i \end{cases}$$

We use the transitive closure of this transition relation, denoted with $\xrightarrow{\mathfrak{ps}}\!\!\!\!\!\rightarrow$, to define reachability.

Definition 6.12 A configuration σ is *reachable*, notation $\text{reachable}(\sigma)$, if and only if there is a sequence of transitions from the empty configuration to σ.

$$\text{reachable}(\sigma) \quad \stackrel{\text{def}}{=} \quad \sigma_\epsilon \xrightarrow{\mathfrak{ps}}\!\!\!\!\!\rightarrow \sigma$$

We prove a lemma stating that any legal configuration is also reachable.

Lemma 6.4 A configuration σ is reachable if it is legal.

$$\text{legal}(\sigma) \implies \text{reachable}(\sigma)$$

Proof.
Let σ be a legal configuration. We build a sequence starting in σ_ϵ leading to σ, using only the second, completely formalized type of transition. Since the routing function has no memory, i.e., is of type $P \times P \mapsto \mathcal{P}(C)$, and since we assume that

all processing nodes can send messages destined for all other processing nodes, each packet f stored in some channel c in σ can be considered as generated by the processing node S at the source of channel c. Therefore, the sequence can be constructed as follows. Let c be some channel not empty in σ. Initially, channel c is empty, since we start with σ_ϵ. All packets assigned to channel c in configuration σ are injected starting with the rightmost packet. With each injection, the transition relation requires 1.) the destination d of the packet to adhere to $\mathbf{R}(S, d) = c$, and 2.) at least one of the places in c to be empty. As the definition of a legal configuration enforces that all packets can be routed towards their current channel, the first requirement is met. As the definition of a legal configuration enforces that at most k packets are assigned to a channel with k places, an empty place always exists. □

Non-canonical deadlocks

Definition 6.7 recognizes canonical deadlocks, i.e., deadlocks in which *all* packets are permanently blocked. Generally, non-canonical deadlocks are of interest as well. We prove that the existence of a non-canonical deadlock is logically equivalent to the existence of a canonical deadlock. Therefore checking only for canonical deadlocks suffices to check for non-canonical deadlocks.

The proof uses the concept of sub-configurations.

Definition 6.13 Configuration σ_S is a *sub-configuration* of σ, notation $\sigma_S \subseteq \sigma$, if and only if for some set of channels its assignment is equivalent to that of σ, and for the remaining channels it is empty.

$$\sigma_S \subseteq \sigma \quad \stackrel{\text{def}}{=} \quad \exists C' \subseteq C \cdot \forall c \in C \cdot \sigma_S(c) = \begin{cases} \sigma(c) & \text{if } c \in C' \\ [\epsilon, \ldots, \epsilon] & \text{if } c \notin C' \end{cases}$$

Definition 6.14 A configuration σ is a *non-canonical deadlock configuration*, notation $\mathbf{\Psi}(\sigma)$, if and only if there exists a sub-configuration that is in deadlock.

$$\begin{aligned} \mathbf{\Psi}(\sigma) \quad \stackrel{\text{def}}{=} \quad & \text{legal}(\sigma) \wedge \\ & \exists \sigma_S \subseteq \sigma \cdot \mathbf{\Omega}(\sigma_S) \end{aligned}$$

Lemma 6.5 There exists a non-canonical deadlock configuration if and only if there exists a canonical deadlock configuration.

$$\exists \sigma \in \Sigma \cdot \mathbf{\Psi}(\sigma) \iff \exists \sigma' \in \Sigma \cdot \mathbf{\Omega}(\sigma')$$

Proof.
($\Longrightarrow$)
Assume a non-canonical deadlock configuration σ. By Definition 6.14, there exists a sub-configuration σ_S that is in deadlock. Configuration σ_S is legal, non-empty, and *all* packets are permanently blocked. Note that Lemma 6.4 ensures that this canonical deadlock is also reachable.
($\Longleftarrow$)
Assume a canonical deadlock configuration σ'. This is a non-canonical deadlock configuration as well. □

Buffers vs. queues

Packet switching allows two types of networks: networks with queues and networks with buffers. One might expect two different definitions of deadlock. Our definition (Definition 6.7) defines deadlock for networks with buffers. A deadlock in a network with buffers requires all packets to be blocked as their next hops are full. In a deadlock in a network with queues, only the packets at the head of the queues are blocked because their next hops are full. These packets subsequently block the packets in the tail of the queue. We prove Lemma 6.6 stating that these definitions are logically equivalent.

Definition 6.15 For packet network N with queues, a configuration σ is a *deadlock configuration*, notation $\mathbf{\Omega}^{\mathrm{q}}(\sigma)$, if and only if it is a legal, non-empty configuration where all packets at the head of the queues are blocked.

$$\begin{aligned}\mathbf{\Omega}^{\mathrm{q}}(\sigma) \quad \stackrel{\text{def}}{=} \quad & \mathrm{legal}(\sigma) \wedge \\ & \sigma \neq \sigma_\epsilon \wedge \\ & \forall c \in C \cdot \sigma(c)[0] \neq \epsilon \implies \\ & \qquad \forall n \in \mathbf{R}(\mathrm{end}(c), \mathrm{dest}(\sigma(c)[0])) \cdot |\sigma(n) - \epsilon| = \mathrm{cap}(n)\end{aligned}$$

We prove that the existence of a deadlock configuration in both types of networks is logically equivalent.

Lemma 6.6 For any network N, there exists a deadlock in N with queues if and only if there exists a deadlock in N with buffers.

$$\exists \sigma \in \Sigma \cdot \mathbf{\Omega}^{\mathrm{q}}(\sigma) \iff \exists \sigma' \in \Sigma \cdot \mathbf{\Omega}(\sigma')$$

Proof.
($\Longrightarrow$)
Assume a configuration σ that is a deadlock in a network with queues. We construct a configuration σ' that is a deadlock in a network with buffers (see Figure 6.7). Replace in σ each packet in the tail of a queue with a copy of the packet at the head of the queue. Configuration σ' is legal, and *all* packets are blocked.

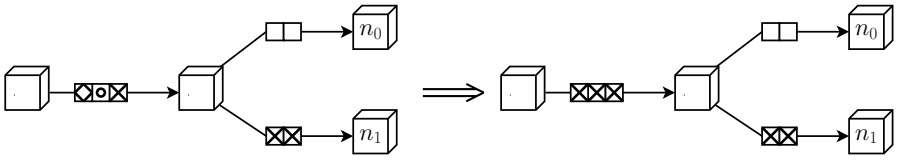


Figure 6.7: *Transformation from a packet network with queues to a packet network with buffers.*

($\Longleftarrow$)
Assume a configuration σ' that is a deadlock in a network with buffers. This configuration is a deadlock in a network with queues as well. □

Lemma 6.6 states that deadlock freedom does not depend on whether channels are implemented with queues or buffers. A necessary and sufficient condition

that applies to networks with buffers also applies to networks with queues. An algorithm that decides deadlock freedom for one type of networks, is also a decision procedure for the other. Since Definition 6.7 involves fewer concepts, we use this as formal definition of deadlock.

6.4 Relation to Duato

There is only one previous necessary and sufficient condition for deadlock-free adaptive routing in packet switching. It has been defined by Duato [45]. We include his condition and a short clarification, but for an extensive explanation we refer to Duato's book [48].

6.4.1 Duato's Condition

> A connected and adaptive routing function R for an interconnection network I is deadlock-free if and only if there exists a routing subfunction R_1 that is connected and has no cycles in its extended dependency graph D_E.

The intuition can be summarized as follows: assume a subgraph C_1 which contains all processing nodes, but contains only a subset of the channels of the network. Let C_1 satisfy two assumptions: (1) C_1 is acyclic and (2) the routing function is able to route any packet to any destination using channels in C_1 only. Then each message will always eventually reach its destination. Even if a message is stuck in a cycle, the channels of this cycle do not belong to C_1 by Assumption (1). Assumption (2) states that for each node, any message can be routed to its destination through channels in C_1. Thus the message can always escape the cycles it is in by using channels in C_1.

Duato formalizes this notion using the concept of *routing subfunction*. Such a function only selects a subset of the possible next hops for each destination. This routing subfunction must be *connected*, i.e., able to route any packet to any destination. Furthermore, he extends the dependency graph with *direct cross dependencies*. If a packet stored in some channel could not have been routed to this channel by routing subfunction R_1, then dependencies involving this channel are direct cross dependencies. The *extended dependency graph* is the dependency graph with added direct cross dependencies.

6.4.2 Relation to our Condition

Our condition is logically equivalent to Duato's one. Both conditions are both necessary and sufficient for deadlock-free routing and the definitions of deadlock are equal. Furthermore, both conditions formalize the same intuition: there must always be an escape. Both conditions isolate the network layer. The data-link layer is abstracted by assuming a message is stuck if and only if all its next hops are unavailable. Assuming that processing nodes can send messages destined for all other processing nodes abstracts away from the application-layer.

The differences lie in the formalization of the intuition. We straightforwardly formulate that there must always exist a packet that is able to escape instead of stating that there must exist a routing subfunction capable to route packets through an acyclic subgraph. Moreover, the use of the regular dependency graph instead of the extended dependency graph reduces the complexity.

The proofs are completely different. We prove the contrapositive form, namely that a deadlock is a subgraph without an escape. This enables a more constructive approach, since we merely had to construct a knot from a deadlock configuration. Duato constructs in his proof an acyclic connected routing subfunction from a network where no deadlock configuration is possible. This is the most difficult part of his proof.

A necessary and sufficient condition for deadlock-freedom of adaptive routing functions in packet networks has been presented. We will now present such a condition for wormhole networks. Some of the definitions can be reused, e.g., the dependency graph. Some – most notably the definitions of deadlock and escape – cannot and will be redefined for wormhole networks.

6.5 Wormhole Switching: Formal Condition

Similar to packet networks, wormhole networks pose constraints on which configurations are legal. Again, no channel capacities may be exceeded. In contrast to packet networks, wormhole switching allows messages to acquire a resource only when it is completely empty. Thus, each channel can contain flits belonging to one message only. Also, while packets occupy one place in some channel, worms occupy paths of channels in the network.

Definition 6.16 In a wormhole network a configuration σ is *legal*, notation $\text{legal}^{\mathfrak{whs}}(\sigma)$, if and only if for any channel the capacity is not exceeded, all flits belong to at most one message, and all flits can be routed towards their current channel.

$$\text{legal}^{\mathfrak{whs}}(\sigma) \quad \stackrel{\text{def}}{=} \quad \forall c \in C \begin{cases} |\sigma(c)| = \text{cap}(c) & (1) \\ |\text{msgs}(\sigma(c))| \leq 1 & (2) \\ \forall m \in \text{msgs}(\sigma(c)) \cdot \mathbf{R}\text{-path}(\text{channels}(m, \sigma)) & (3) \end{cases}$$

If it is clear from the context that the definition concerns wormhole networks, the superscript $\mathfrak{whs}$ will be omitted.

A deadlock configuration is required to be legal. Just as for packet networks, it must be non-empty. Additionally, all header flits are blocked. A header flit is blocked if all its next hops contain at least one flit. As will be shown in Section 6.6, there is no need for tail flits to be blocked as well. We assume predicates hd and tl are available which return true if and only if the given flit is a header (tail) flit. The following three properties make a legal configuration a deadlock:

4. It is not empty.
5. No header flit has arrived at its destination.

6. All header flits are blocked.

Definition 6.17 For wormhole network N, a configuration σ is a deadlock configuration, notation $\mathbf{\Omega}^{\mathfrak{whs}}(\sigma)$, if and only if it is legal, non-empty and satisfies the following properties.

$$\mathbf{\Omega}^{\mathfrak{whs}}(\sigma) \stackrel{\text{def}}{=}$$

$$\begin{array}{ll} \text{legal}(\sigma) \wedge & (1\text{–}3) \\ \sigma \neq \sigma_\epsilon \wedge & (4) \\ \forall c \in C \cdot \forall f \in \sigma(c) - \epsilon \cdot \text{hd}(f) \implies & \\ \qquad \left\{ \begin{array}{ll} \text{dest}(f) \neq \text{end}(c) & (5) \\ \forall n \in \mathbf{R}(\text{end}(c), \text{dest}(f)) \cdot |\sigma(n) - \epsilon| > 0 & (6) \end{array} \right. & \end{array}$$

6.5.1 Our Condition

There are two major complications inherent to wormhole switching with respect to packet networks. First, for packet switching any subgraph can be filled with packets in such a way that a legal configuration is obtained. In wormhole networks, only subgraphs that can be filled with pairwise disjoint worms can yield a legal configuration. Secondly, in packet networks any packet can be routed autonomously towards its next hops. In wormhole networks, the header flit moves autonomously, but the tail flits always follow the header flit.

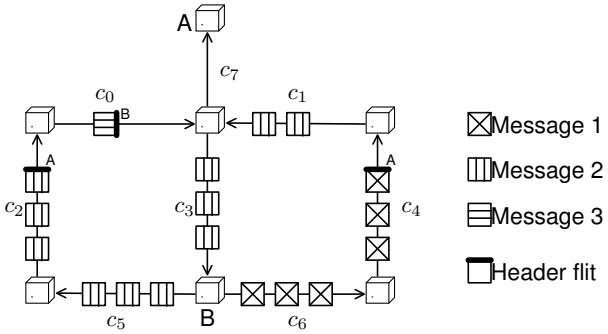
To illustrate the first issue, Figure 6.8 shows a legal and an illegal deadlock configuration. The legal configuration contains three messages. Message 1 is destined for node A. It is blocked by Message 2 that holds channel c_1. This channel is the only channel leading from channel c_4 to node A. Message 2, destined for node A as well, is blocked by Message 3. Message 3, consisting of one flit only, is blocked by Message 2.

The illegal configuration violates the property that in wormhole networks a channel contains flits of at most one message. Channel c_3 contains flits of Messages 1 and 2. In a legal configuration, the worms alwys constitute a pairwise disjoint set of routing paths. Our condition requires a pairwise disjoint set of paths to form a deadlock.

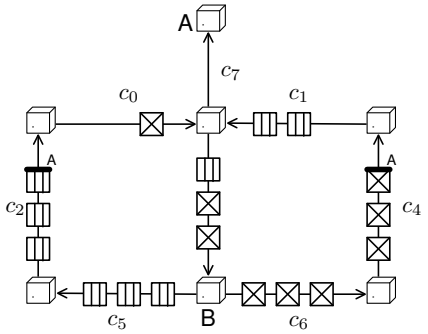
To illustrate the second issue, Figure 6.9 shows a legal configuration that is not in deadlock. As in Figure 6.8a there are three messages. Messages 2 and 3 are blocked. Message 1 is not blocked, as it can now use the new channel c_8 to advance towards its destination. It can *escape* the congested area. Our condition requires the absence of escapes to form a deadlock.

Channel c_8 does not necessarily provide an escape. First, channel c_4 must be filled with a header flit. Only a header flit can escape, as tail flits follow the header flit and cannot use the escape. For example, Figure 6.8a shows a deadlock configuration, even though there is an escape for the tail flits in channel c_1. Secondly, the escape must be supplied by the routing function for the destination of the worm. Consider the network in Figure 6.9. If channel c_8 is not supplied for destination A, it cannot be used as an escape for Message 1. We need to redefine the notion of escape for wormhole networks.

Definition 6.18 Given a set of routing paths $\Pi^\star$, a channel e is an escape for $\Pi^\star$, notation $\text{esc}(e, \Pi^\star)$, if and only if channel e is the head of some path and if



(a) *Legal deadlock configuration*



(b) *Illegal deadlock configuration*

Figure 6.8: *Legal and illegal deadlock configurations.*

either e is a sink or e has a dependency neighbor that is not contained in $\Pi^\star$.

$$\mathrm{esc}(e, \Pi^\star) \stackrel{\text{def}}{=} \exists \pi^d \in \Pi^\star \cdot e = \pi^d[0] \wedge (A_{\mathtt{dep}}(e, d) = \emptyset \vee A_{\mathtt{dep}}(e, d) \not\subseteq \bigsqcup \Pi^\star)$$

Our condition states that the absence of escapes for some pairwise disjoint set of routing paths is necessary and sufficient to create a deadlock, or contrapositively:

Theorem 6.2 A wormhole network is deadlock-free if and only if all pairwise disjoint sets of routing paths have an escape.

$$\mathsf{DLF}(N) \iff \forall \Pi^\star \in \mathcal{P}(\mathcal{L}(C)) \cdot \left\{ \begin{array}{l} \Pi^\star \neq \emptyset \\ \bigsqcap \Pi^\star = \emptyset \\ \mathbf{R}\text{-paths}(\Pi^\star) \end{array} \right. \implies \exists e \in C \cdot \mathrm{esc}(e, \Pi^\star)$$

We provide two examples: one with a deadlock and one where the network is deadlock-free.

Example 6.3 Consider the interconnection network in Figure 6.10a without the dashed channel G. The network consists of five processing nodes. For sake of clarity, only the processing nodes d_0 and d_1 are destination nodes. The routing function is depicted in Figure 6.10b. There is exactly one deadlock possible. This deadlock has two worms. One worm occupies channels A and B. Its tail is in A and its header flit is in B. One worm occupies channel C only and is destined

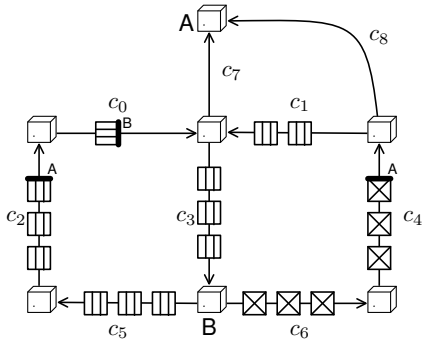


Figure 6.9: *A legal configuration that is not in deadlock.*

for d_0. The worms wait for each other and have no alternative routes towards their destinations. The circular wait is represented by the cycle in the dependency graph in 6.10c. The set of paths $\{[B, A], [C]\}$ has no escape. There is a deadlock.

Example 6.4 Consider again the interconnection network in Figure 6.10a, including the dashed channel G. Routing is extended with $\mathbf{R}(n_2, d_0) = G$. We prove the network deadlock-free with our condition, by considering all pairwise disjoint sets of routing paths. Set $\{[B, A], [C]\}$ is such a set. Channel C is an escape, as it is a channel at the head of one of the paths and it provides an alternative route for both destinations. Set $\{[A], [B], [C]\}$ is another set. Again, channel C is an escape. There are similar sets of paths. In each pairwise disjoint set of routing paths, either channel A or channel C is an escape. There is no deadlock.

6.5.2 Proof

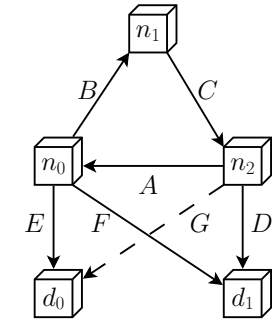
We prove the contrapositive version of Theorem 6.2. We split up the proof into two lemmas. Lemma 6.7 proves that our condition is sufficient for deadlock freedom. The intuition is that from a deadlock we construct a witness, i.e., a set of routing paths that falsifies our condition. Lemma 6.8 proves necessity of our condition. The proof is again constructive: we build a witness, i.e., a deadlock, from a set of routing paths.

Lemma 6.7 There is a pairwise disjoint set of routing paths without an escape if there is a deadlock configuration.

Proof. Assume a deadlock configuration σ. We construct a witness $\Pi^\star$:

$$\Pi^\star \stackrel{\text{def}}{=} \{\text{channels}(m, \sigma) \mid \exists c \in C \cdot m \in \text{msgs}(\sigma(c))\}$$

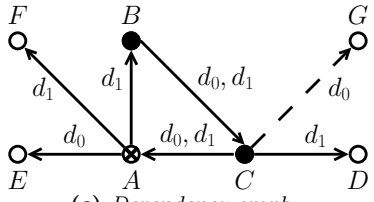
We show that this witness falsifies our condition. We prove that is satisfies the three properties on the left hand side of the implication in Theorem 6.2, but that it does not satisfy the right hand side. A deadlock configuration is by Property (4) non-empty and therefore $\Pi^\star$ contains at least one path. Property (2) of Definition 6.17 ensures that channels contain flits of at most one message. This means that all paths in $\Pi^\star$ are pairwise disjoint. Property (3) of Definition 6.17 ensures



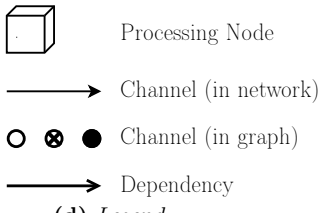
(a) *Interconnection network*

n	$\mathbf{R}(n, d_0)$	$\mathbf{R}(n, d_1)$
n_0	E	B, F
n_1	C	C
n_2	A	A, D

(b) *Routing Function*



(c) *Dependency graph*



(d) *Legend*

Figure 6.10: *Example of deadlock in a wormhole network. The dashed channels are used in Example 6.4 only.*

that the worms constitute routing paths. The left hand side of the implication is satisfied. We now prove $\Pi^\star$ has no escape. Property (5) of Definition 6.17 states that no header has reached its destination. Hence, the head of each routing path in $\Pi^\star$ has at least one dependency neighbor for its destination. Property (6) of Definition 6.17 states that all next hops of all header flits are unavailable, i.e., they contain at least one flit. Thus, these next hops are part of some routing path. This shows that for any routing path in $\Pi^\star$ the set of dependency neighbors of its head for its destination is a subset of the union of $\Pi^\star$. A deadlock implies a set of pairwise disjoint paths without an escape. □

Lemma 6.8 There is a deadlock configuration if there is a pairwise disjoint set of routing paths without an escape.

Proof. Assume a pairwise disjoint set of paths $\Pi^\star$. We build a witness deadlock configuration σ by building a list of messages M. For each path π^d in $\Pi^\star$, there is a message with destination d. This message fills each channel in π^d with one flit. We show that σ satisfies the six properties of a deadlock configuration:

(1) As the paths are pairwise disjoint, each channel contains at most one flit. No buffer capacity in σ is exceeded. For each channel c in each routing path π^d, destination d is in the typing information of channel c.

(2) As the paths are pairwise disjoint, each channel is filled with flits belonging to one message only.

(3) As the worms are built from routing paths, they are valid worms.

(4) As there is at least one routing path, there is at least one worm.

(5) As the head of each path has at least one dependency neighbor, no message arrives at its destination.

(6) As all d-neighbors are included in the union of $\Pi^\star$ and as each channel in each path in $\Pi^\star$ is filled, no header flit has an available next hop.

□

Theorem 6.2 follows directly from Lemmas 6.7 and 6.8.

6.6 Definition of Deadlock

Definition 6.17 is justified by lemma's similar to Lemma's 6.4 and 6.5 (respectively [45] and [48]), which state that any legal configuration is reachable, and that is suffices to check for canonical deadlocks. Additionally, we prove Lemma 6.9 that states that adding a case distinction for tail flits is superfluous. Wormhole networks have tail and header flits. Header flits are blocked by their next hops. Tail flits are blocked if the channel occupied by the next flit in the worm is full. It is possible that the header flit of a worm is blocked, but tail flits can still move. Since both are blocked under different conditions, one might expect that Definition 6.17 includes a case distinction for both types of flits. The extra case distinction for tail flits is formalized as follows. We assume that given a tail flit of a worm, it is possible to compute the channel occupied by the next flit in the worm. Function next : $F \times \Sigma \mapsto C$ returns the next channel of a tail flit given the current configuration.

Definition 6.19 For wormhole network N, a configuration σ is a deadlock configuration with blocked tail flits, notation $\mathbf{\Omega}^{(1\text{–}7)}(\sigma)$, if and only if it is legal, non-empty and satisfies the following properties.

$$\mathbf{\Omega}^{(1\text{–}7)}(\sigma) \stackrel{\text{def}}{=}$$

$$\begin{array}{ll} \mathbf{\Omega}(\sigma) & (1\text{–}6) \\ \forall c \in C \cdot \forall f \in \sigma(c) \cdot \mathrm{tl}(f) \implies |\sigma(\mathrm{next}(f)) - \epsilon| = \mathrm{cap}(\mathrm{next}(f))) & (7) \end{array}$$

From a configuration where tail flits can still advance, it is possible to construct a deadlock configuration with blocked tail flits that satisfies all seven properties.

Lemma 6.9 There exists a deadlock configuration with blocked tail flits if and only if there exists a configuration which satisfies Definition 6.17.

$$\exists \sigma \in \Sigma \cdot \mathbf{\Omega}^{(1\text{–}7)}(\sigma) \quad \Longleftrightarrow \quad \exists \sigma' \in \Sigma \cdot \mathbf{\Omega}(\sigma')$$

Proof.
($\Longrightarrow$)
Take $\sigma' = \sigma$. Since Properties (1) to (7) hold for σ, Properties (1) to (6) hold for σ'.
($\Longleftarrow$)

Consider a configuration σ' that satisfies Properties (1) to (6). We show there exists a legal deadlock configuration σ. Construct σ by filling all channels with the exact same worms as in σ', but with all channels filled completely. Thus some worms in σ may consist of more flits than they originally consisted of in σ'. Since σ' satisfies Properties (1) to (6), configuration σ does as well. Furthermore, configuration σ' satisfies Property (7) as all resources are filled completely. □

6.7 Relation to Duato

Duato defined a necessary and sufficient condition for adaptive deadlock-free routing. This condition is presented, together with a counterexample showing that Duato's condition is not necessary and sufficient. We will show that our theorem subsumes Duato's.

6.7.1 Duato's Condition

Duato formalized the notion of escapes using the concept of a *routing subfunction*. Duato proved that if it is possible to restrict a routing function in such a way that the corresponding dependency graph – called the *extended* dependency graph – becomes acyclic, the routing function is deadlock-free. The resulting routing subfunction must still be able to route any message to any destination, i.e., it must still be *connected*.

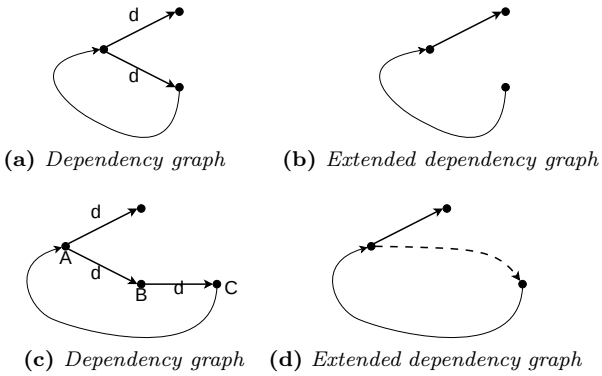
> An interconnection network with adaptive routing function R is deadlock-free if there exists a connected routing subfunction R_1 with an acyclic extended dependency graph.

Consider the channel dependency graph in Figure 6.11a. The dependency graph contains a cycle. This indicates the existence of a circular wait in the network. There is an adaptive point where a message can either escape the cycle or be routed into the dependency cycle. At this point the circular wait can be resolved, as a message can escape the cycle. The cycle is not sufficient to create a deadlock.

It is possible to restrict the routing function in such a way that the extended dependency graph becomes acyclic. Namely, if the routing subfunction supplies the escape channel only and restricts the use of the channel leading into the cycle. Figure 6.11b gives the corresponding acyclic extended dependency graph. If there exists a connected routing subfunction with an acyclic extended dependency graph, the original routing function is deadlock-free.

When the routing subfunction restricts the use of a channel, this does not mean that the channel is not used by the original routing function. Consider Figure 6.11c. Say the routing subfunction restricts the use of channel B, making the extended dependency graph acyclic. However, as channel B is supplied by the original routing function, a worm might occupy channels A and B simultaneously. Then the escape for channel A cannot be used as the tail flits in channel A follow the header flit in channel B. The cycle must have another escape.

In this case, progression of the message in channel A depends on the message in channel C. The extended dependency graph must reflect this dependency. Duato



(a) *Dependency graph* **(b)** *Extended dependency graph*

(c) *Dependency graph* **(d)** *Extended dependency graph*

Figure 6.11: *Extended dependency graphs*

introduces *indirect dependencies*. If there is a routing path of channels not supplied by the routing subfunction, there is an indirect edge (see Figure 6.11d).

Duato's theorem involves two other types of edges: *cross direct* and *cross indirect* edges. As we do not require them in this Section, we will provide no further details. For a more extensive introduction to Duato's theorem we refer to Duato's papers [48, 44].

Duato's theorem holds only for *coherent* routing functions. Let π be some path from channel A to B that can be established by the routing function for some destination d. A routing function is coherent if and only if any subpath of π from channel A to an intermediate channel B' can be established by the routing function for destination d' as well, where d' is the processing node at the end of channel B'. Duato provides an example of an incoherent deadlock-free routing function that cannot be proven deadlock-free with his theorem [44]. We provide this example.

See Figure 6.12 for the interconnection network. The routing function is defined as follows: if destination j is higher than current node i, use c_{Hi} or c_{A1} if $i = 1$ or c_{B2} if $i = 2$. For $j < i$, use c_{Li}. This routing function is incoherent. There is a path $[c_{H2}, c_{A1}, c_{B2}, c_{H1}]$ supplied for destination 3 (note that the first channel in the path is its head). However, the subpath $[c_{B2}, c_{H1}]$ is not supplied for destination 1. The network is deadlock-free but cannot be proven deadlock-free with Duato's theorem.

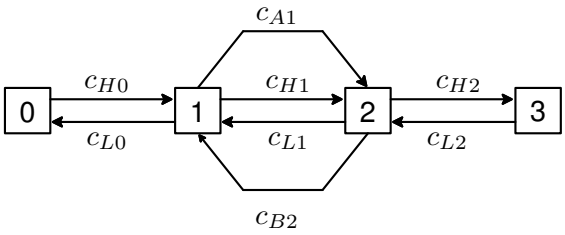


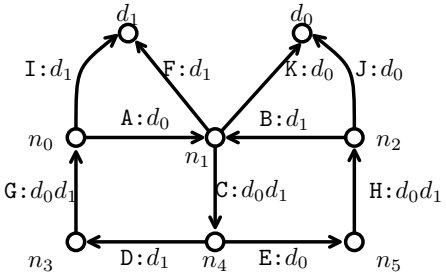
Figure 6.12: *Example of incoherent routing function [44]*

6.7.2 A Counterexample

We show our counterexample. We then point to the root cause for this counterexample and propose a fix to keep the condition necessary.

Consider the interconnection network in Figure 6.13. It has processing nodes n_i $(0 \leq i \leq 5)$, d_0 and d_1. Figure 6.13 only shows the channels – A to K – relevant to our counterexample. For all pairs of processing nodes (n_i, n_j) $(i \neq j)$, there is a dedicated channel $\mathcal{D}_{ij}$ from n_i to n_j. Also, there are dedicated channels from d_0 to all other processing nodes. The same holds for d_1. Since these dedicated channels lead directly to their destination, they are empty in any canonical deadlock configuration. Any such deadlock configuration consists of messages created in some node n_i and destined for some node d_i. Our counterexample considers those messages only. For sake of clarity we do not mention the dedicated channels any further.

The routing function is specified by the table in Figure 6.13 for destinations d_0 and d_1. For any $s = n_i$ and $d = n_j$ $(i \neq j)$, let $\mathbf{R}(s, d)$ supply – besides the dedicated channels – all channels that start a shortest path from n_i to n_j. For instance, $\mathbf{R}(s, d)$ supplies both the dedicated channel $\mathcal{D}_{05}$ and channel A. This makes the routing function coherent.



R	d_0	d_1
n_0	A	I
n_1	{C, K}	{C, F}
n_2	J	B
n_3	G	G
n_4	E	D
n_5	H	H

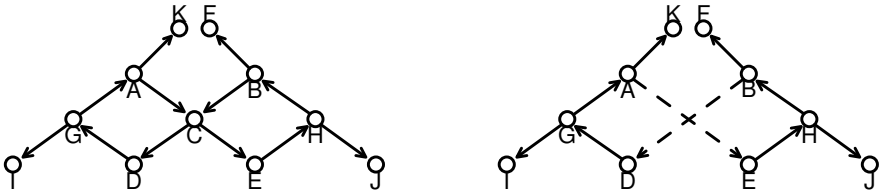
Figure 6.13: *The interconnection network and the routing function*

This network is deadlock-free. This can be seen as follows. Channels I, F, K and J have to be empty. Otherwise a message can arrive at its destination. If there is a header flit in either channel A or B, this header flit can escape to channel F or K and arrive at its destination. If channels A and B both contain tail flits only, at least one of the channels F or K contains a header flit, implying a message arrives at its destination. If either channel A or B is filled with tail flits only and the other is empty, there is always a header flit that can move. Say channel A has tail flits only and channel B is empty. There is a header flit in either C, E or H that can move forward. Say channel B has tail flits only and channel A is empty. There is a header flit in either C, D or G that can move forward. No deadlock is possible.

Duato's condition states that a network is deadlock-free if and only if there exists a routing subfunction that is connected and has no cycles in its *extended* dependency graph. For our purpose it is enough to define this graph as being the dependency graph extended with *indirect* dependencies. We now show that there is no such routing subfunction for the network defined in Figure 6.13.

We have to consider all possible connected routing subfunctions $\mathbf{R}_1$ and show

that the corresponding extended dependency graph contains a cycle. As Figure 6.13 shows, processing node n_1 is the only adaptive point. Restricting the routing functions at other points than n_1 makes the routing subfunction disconnected. Regarding processing node n_1, restricting the use of channels F and K makes no sense, as this cannot possibly make the dependency graph acyclic. Thus, there are three connected routing subfunctions: restrict the use of channel C completely or restrict the use of C for one of the destinations d_0 or d_1.



(a) *Dependency graph* **(b)** *Extended dependency graph*

Figure 6.14: *The dependency graph and the extended dependency graph corresponding to the routing subfunction restricting channel C completely. The dotted arrows denote a indirect edge.*

Figure 6.14a shows the dependency graph of the network. We consider the case where routing is restricted such that channel C is not used for any destination. Edges (A, C), (B, C), (C, D) and (C, E) can be removed from the dependency graph. The current routing subfunction creates two indirect edges: (A, E) and (B, D). Let us consider edge (A, E). There is a path from $\text{src}(A) = n_0$ to $\text{end}(E) = n_5$ for messages destined for d_0 such that A and E are respectively the first and last channels of this path and the only ones supplied by $\mathbf{R}_1$. Channel C, the only intermediate channel, is not supplied by $\mathbf{R}_1$. Analogously for edge (B, D). The extended dependency graph is not acyclic. It contains cycle $[A, G, D, B, H, E, A]$ (Figure 6.14b).

The other two routing subfunctions, created by restricting the use of channel C for one of the destinations d_0 or d_1, are cyclic for the exact same reason. As all routing subfunctions have cyclic extended dependency graphs, Duato's condition states that the network has a deadlock, whereas it is deadlock-free.

The network could be in deadlock, if worms could intersect. Consider the following – illegal – configuration. Channels G and H are filled with messages destined for d_0 and d_1 respectively. There are two intersecting worms, one occupying channels A, C, and E (destined for d_0) and one occupying B, C and D (destined for d_1). None of these messages can move. There is a deadlock.

In wormhole networks, channels can contain flits belonging to one message only. Duato states this in Assumption 5 of his article [44]. Duato mentions this assumption in his informal definition of a legal configuration. However, he omits this in his formal definition of a legal configuration. The consequence is that when proving the condition – Theorem 4, parts b2a) and b2b2) – a configuration is built by filling indirect edges with flits belonging to one message. By definition, information on which channels are used in an indirect edge are lost. To build a true configuration one has to prove that these edges do not intersect. Otherwise, Assumption 5 may be violated and the deadlock configuration is not legal.

We suggest a possible fix for this issue. A legal cycle is defined as a cycle in the extended dependency graph such that the (cross) indirect edges can constitute paths of channels that are pairwise disjoint. The fixed condition would become:

> A coherent, connected and adaptive routing function R for an interconnection network is deadlock-free if and only if there exists a routing subfunction R_1 that is connected and has no legal cycle in its extended dependency graph.

The proof of this Theorem is an exact copy of Duato's proof, with a small extension in parts b2a) and b2b2) to deal with the new definition of a legal configuration.

The discrepancy is caused by an omission in the *formalization* of the definition of deadlock. We did not find any inconsistency in Duato's informal definitions or in his complex proof. In Chapter 7, we will show that this minor discrepancy makes deciding deadlock freedom on wormhole networks co-NP-complete, instead of polynomial.

6.7.3 Relation to our Condition

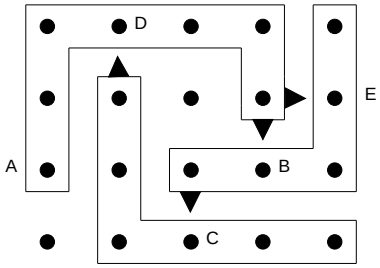
We will show that if Theorem 6.2 identifies a deadlock, Duato's condition does as well. The other direction cannot be shown.

Assume a set S which is the union of a set of routing paths for which no head has an escape. Figure 6.15a gives an example. We show that any connected routing subfunction R_1 has a cyclic extended dependency graph. First we show this for the unrestricted routing function R. Let π be the routing path for destination d starting in channel A. Channel B is a d-neighbor of the head of π. In the dependency graph there is a path of dependencies from channel A to B. As S has no escape, channel B is again member of some routing path for some destination d'. Thus there is another path of dependencies from B to a channel C in another routing path for some destination d'', and so on. As S has no escape, the paths of dependencies will eventually lead into one or more cycles, see Figure 6.15b.

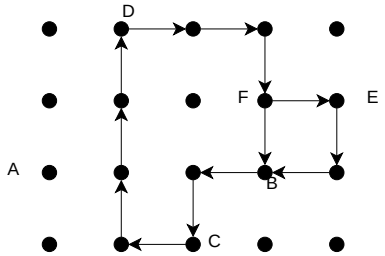
These cycles cannot be broken by restricting the routing function. The routing subfunction can be restricted either 1) inside some routing path, 2) at the head of a routing path, or 3) none of these.

First, say the routing subfunction restricts routing inside a routing path. This may break such a path into pieces. However, since a routing path consists of dependencies created by the same destination only, any hole in the path will be bridged by an *indirect* edge. Thus the cycle remains intact. Figure 6.15c shows an indirect edge in a routing path. Secondly, say the routing subfunction is restricted at the head of some routing path. E.g., say the routing subfunction restricts routing in channel F to use channel E only. This breaks dependency (F, B) but does not break all cycles. As S has no escape *all* dependencies at head F caused by destination d lead back to S. This means that to break all cycles, all these dependencies must be broken. In the example this means that both dependencies (F, E) and (F, B) must be broken to break all cycles. This is not possible as the routing subfunction must be connected.

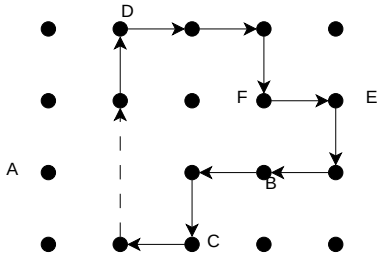
Lastly, restricting the routing function outside of the routing paths will not break any of the dependency cycles.



(a) *A set of routing paths in the dependency graph. The arrows point to the dependency neighbors caused by the destination of the routing path of the head of each path.*



(b) *The related cyclic edges in the dependency graph.*



(c) *The cyclic edges in an extended dependency graph. The dotted line is an indirect edge.*

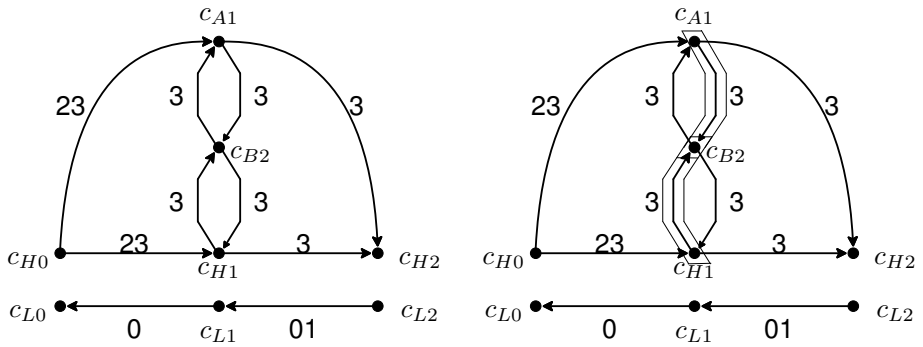
Figure 6.15: *A set of routing paths related to Duato's condition.*

Assuming the existence of a set of routing paths without an escape, there is no connected routing subfunction with an acyclic extended dependency graph. The other direction does not hold necessarily. A cycle in the extended dependency graph can be translated to a set of routing paths, but as it is unknown what channels are needed to fill an indirect dependency, the paths are not necessarily pairwise disjoint.

In contrast to Duato's theorem, our condition holds for incoherent routing functions as well. We prove the example in Figure 6.12 deadlock-free. Figure 6.16a shows the dependency graph. The edges are labeled with the destinations causing the dependencies. We show that there is no pairwise disjoint set of paths where no header has an escape. Assume such a set of paths. We proceed by contradiction.

We first show that channel c_{B2} must be the head of a path and that channels c_{A1} and c_{H1} must be included in the tails of some paths. Channel c_{H2} must be empty as it has no neighbors. Channels c_{A1} and c_{H1} cannot be heads as otherwise channel c_{H2} is an escape. Thus there are only two possible channels which can be at the head of a path: c_{H0} and c_{B2}. Channels c_{A1} and c_{H1} must be included in the tails of some paths since otherwise they would be escapes for both c_{H0} and c_{B2}. Channel c_{B2} must be the head of a path since otherwise the paths containing c_{A1} and c_{B2} have no heads.

Thus channel c_{B2} is the head of a path and channels c_{A1} and c_{H1} are included in



(a) *Dependency graph* **(b)** *Two paths where no head has an escape.*

Figure 6.16: *Dependency graph of network in Figure 6.12*

the tails of some paths. Here we have arrived at a contradiction. Channel c_{B2} can be the head of at most one path. This path must contain both c_{A1} and c_{H1}. This cannot be done without the worm intersecting itself. E.g., path $[c_{B2}c_{H1}c_{B2}c_{A1}]$ is a path with its head in c_{B2} that contains channels c_{A1} and c_{H1}. However, this path intersects itself. There is no pairwise disjoint set of paths where no head has an escape. By Theorem 6.2 the network is deadlock-free.

Note that there exists a set of routing paths where no header has an escape. Consider the set of paths $\{[c_{B2}, c_{A1}]\ ,\ [c_{B2}, c_{H1}]\}$, both drawn in Figure 6.16b. For this set of paths no head has an escape. It is however not a pairwise disjoint set of paths.

6.8 Relation to Schwiebert and Jayasimha

6.8.1 Schwiebert and Jayasimha's Condition

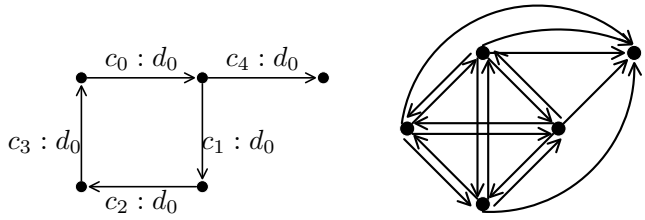
Schwiebert and Jayasimha defined a necessary and sufficient condition for deadlock-free routing.

> A routing function R is deadlock-free if and only if R is wait-connected for some subgraph S and S has no True Cycles in the waiting graph.

Schwiebert and Jayasimha's condition depends on the *waiting graph*. The waiting graph has as vertices the channels of the network. There is an edge between two channels A and B if B is a *waiting channel* for A, i.e., if a blocked message in A can wait for channel B to become available. Channels A and B do not necessarily have to be topological neighbors: if a worm occupies multiple channels, among which channel A, and waits for channel B then B is a waiting neighbor of A.

Consider the network in Figure 6.17a. Channels c_0 to c_3 all are waiting neighbors of each other. Channel c_4 is a waiting neighbor of channels c_0 to c_3. Figure 6.17b gives the corresponding waiting graph.

To discharge Schwiebert and Jayasimha's condition, one needs to selectively remove waiting edges until an acyclic subgraph is obtained. The subgraph must



(a) *Network* **(b)** *Waiting graph*

Figure 6.17: *Example of the waiting graph*

remain *wait-connected*, i.e., each channel must have at least one waiting channel. If there is an acyclic wait-connected subgraph, the routing function is deadlock-free.

An edge in the waiting graph may span multiple channels. Given a cycle of waiting edges, there might not be a legal configuration filling all channels corresponding to the waiting edges. For example, two edges may share a channel, in which case they can only be filled by intersecting worms. Cycles which can(not) be filled are called *True Cycles* (False Resource Cycles). A network is deadlock-free if and only if there is a wait-connected subgraph S that contains no True Cycles.

6.8.2 Relation to our Condition

Schwiebert and Jayasimha's condition is a necessary and sufficient condition for deadlock-free routing. It is defined for a broad class of routing functions. The routing function need not to be memoryless, i.e., routing functions of type $C \times P \mapsto C$ are supported. Also, it allows routing functions to make a distinction between blocked messages and messages that are not blocked. Such routing functions allow more flexibility when messages are not blocked, which can prevent deadlocks.

However, Schwiebert and Jayasimha's condition is a dynamic condition. It does not only depend on a static graph, but also on configurations. In order to determine whether a cycle in the waiting graph is a True Cycle, it must be determined whether there is some reachable configuration that fills the cycle. Note that since routing is not necessarily memoryless, a legal configuration is not necessarily reachable. Thus analysis of injection sequences of messages is required to distinguish True Cycles from False Resource Cycles [48]. Schwiebert and Jayasimha provide an algorithm which performs this analysis, but this algorithm is exponential [128].

As their condition is dynamic and defined for different types of routing functions, we do not further relate this condition to ours.

6.9 Relation to Taktak et al.

6.9.1 Taktak's Condition

Taktak et al. defined a sufficient condition for deadlock-free routing [138]. Furthermore, they created an algorithm checking this condition in polynomial time. Their condition depends on a labeled dependency graph. A channel is labeled with destination d if a message destined for d can occupy the channel. Taktak et al.

define a tagging condition and prove that if all labels can be tagged, the network is deadlock-free.

Definition 6.20 A label l of a node v is tagged if and only if for all successors of v which owns l as label, l is tagged, and there is at least one successor of v having all its labels tagged.

> An interconnection network is deadlock-free if there is at least one label of one channel that cannot be tagged.

Consider the cycle in Figure 6.18. Say we want to determine whether label l_0 can be tagged for channel A. This recursively depends on the next channel on the cycle. During this process visited channels are assumed to be untagged. Eventually, label l_1 must be tagged for channel B. In order to tag label l_1 for channel B a neighbor is required which has all its labels tagged. This cannot be neighbor A, as A has already been visited and thus label l_0 is considered untagged. The only way to tag label l_1 for channel B is to have an *unvisited* escape channel C where all labels, including label l_1, can be tagged.

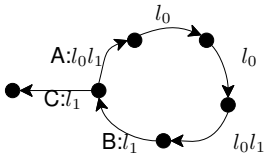


Figure 6.18: *Example of cyclic tagging*

If C again leads into a cycle, than this cycle must again have an unvisited escape in order to get labels tagged. In a deadlock-free network, all cycles will eventually be escaped and all labels will be tagged.

6.9.2 Relation to our Condition

We show that any deadlock identified by Theorem 6.2 is identified as a deadlock by the condition of Taktak et al. The reverse does not hold, as the condition of Taktak et al. identifies false deadlocks. The counterexample presented in Section 6.7.2 also applies to their condition.

Consider again the pairwise disjoint set of routing paths in Figure 6.15a. Let the tagging start in channel A. Tagging of A recursively depends on the next channel in the path. Arriving at channel F, both channel B and E must be tagged. The tagging of B eventually leads to D and thus into a cycle. This cycle has an unvisited escape, namely channel E. This path also leads into a cycle. Since S has no escape, eventually all paths will lead into a cycle and thus eventually there will be no unvisited escape. The result is that for each channel in a routing path for destination d, this destination cannot be tagged. Any deadlock identified by Theorem 6.2 is identified as a deadlock by Taktak et al.

6.10 Conclusion

We have presented necessary and sufficient conditions for deadlock freedom of packet and wormhole networks. Using the notion of escapes, we could define simpler conditions than previous ones. They involve fewer concepts and require fewer definitions. Most notably, the conditions have been defined using only the regular dependency graph. As this graph can be statically computed from a specification of the routing function and the network topology, our conditions are static. Our condition has fewer assumptions, widening its applicability. It can be applied to routing functions which can send messages into a cycle. Effectively, this separates the proof of deadlock freedom from the proof of livelock freedom. Our conditions have been shown to subsume all previous conditions.

To ensure correctness of all definitions and proofs, we have mechanically proven our conditions correct using the GeNoC framework and the ACL2 theorem prover. This mechanical proof has demonstrated its value, as we have found discrepancies in previous work. In all previous work, the definition of a deadlock in wormhole networks has been incorrect. We have supplied a counterexample showing that Duato's condition is only sufficient. The issue is subtle but essential: worms necessarily do not intersect. As a result, our condition for wormhole networks is the first static necessary and sufficient condition for deadlock freedom.

Still, discharging these conditions for an actual chip design is a non-trivial and cumbersome task. In the next chapter, we present algorithms that given a specification of the network automatically check whether these conditions hold.

Chapter 7
Deadlock Detection Algorithms

The previous chapter presented theorems for deadlock-free routing. These theorems provide exact conditions under which a network is deadlock-free. This chapter presents a *quick* and fully *automated* approach to determine that a network actually satisfies these conditions.

We present two algorithms that take as input a specification of the topology and the routing function (see Figure 7.1). In case of a deadlock, the algorithms provide detailed and accurate feedback on the cause of this deadlock. The correctness of our algorithms has been mechanically verified using the ACL2 theorem prover.

For packet networks, our algorithm decides deadlock freedom in linear time with respect to the size of the dependency graph. One might expect that a similar result is feasible for wormhole networks. However, we prove that for wormhole networks deciding deadlock freedom is co-NP-complete. We point to the cause of this complexity and formulate a notion of deadlock freedom that is decidable in polynomial time. The wormhole algorithm checks this sufficient condition, which means that if it returns a deadlock-free result, this result is sound. However, if it returns a deadlock, this is not necessarily a legal and reachable deadlock.

The first part of this chapter deals with packet networks. The algorithm is explained in detail using examples from the previous chapter. Pseudo code is presented of which correctness and algorithmic complexity is proven. The second part deals with wormhole networks. The algorithm is very similar but has to deal with the many subtleties of wormhole switching. In the presentation of this algorithm we focus mainly on the differences between the two algorithms.

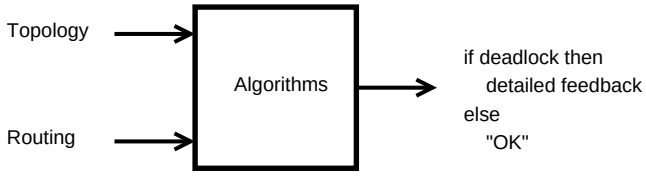


Figure 7.1: *Our approach for determining deadlock freedom.*

7.1 Packet Switching: Algorithm by Example

The basic objective of the algorithm is to mark each channel as *deadlock-immune* or *deadlock-sensitive*. The intuition behind these markings is that in a deadlock-immune channel no packet can be permanently blocked, whereas in a deadlock-sensitive channel some packet can possibly be permanently blocked. After termination of the algorithm, the markings are used to either output the exact reason for deadlock or to state that the network is deadlock-free.

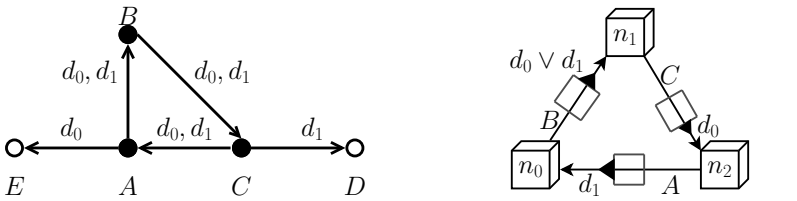
In order to determine the marking of a channel, the algorithm classifies all destinations in the typing information of that channel. A classification $x \mid y$ of channel c stands respectively for destinations that may lead packets from channel c into a circular wait and for destinations for which channel c is an escape. As routing is adaptive, a destination may be classified as both. If any destination that leads packets into circular waits also provides escapes out of these cycles, the channel will be marked deadlock-immune. Otherwise, it will get the marking deadlock-sensitive.

We first demonstrate by example the notions of deadlock-immunity and sensitivity. Secondly we provide an example trace of the algorithm, showing how it detects a deadlock. Lastly, we provide an example trace that reveals the necessity of a post-processing step.

7.1.1 Deadlock-immunity and -sensitivity

We recapitulate the deadlock in the network of Example 6.1. Figure 7.2a shows the dependency graph. Subgraph $\{A, B, C\}$ can form a deadlock. For channel A destination d_1 leads into the subgraph. As there is a destination that does not lead out of the subgraph, channel A is not an escape. The same holds for channel C and destination d_0. Subgraph $\{A, B, C\}$ is a subgraph without an escape.

Figure 7.3h shows the result of our algorithm. The channels A, B and C have been marked deadlock-sensitive. The other channels are deadlock-immune. For channel A destination d_1 is classified as a destination that leads into a circular wait, but that does not provide an escape. Again, the same thing holds for channel C and destination d_0. Figure 7.2b shows the deadlock that is represented by these markings.



(a) *Dependency graph* **(b)** *Deadlock returned by algorithm*

Figure 7.2: *Example dependency graph. The black channels can form a deadlock.*

The algorithm determines for each channel c a classification $x \mid y$ of all destinations in the typing information of c, i.e., for all reachable destinations. If x

is not a subset of y, the channel is marked deadlock-sensitive. Otherwise it is deadlock-immune.

For example, we derive the marking of channel C. Destination d_1 is a type of channel C. As it leads to a sink (channel D) it is classified as a destination for which there is an escape. But as destination d_1 can also lead packets to channel A, it is also classified as a destination that may lead packets to a circular wait. Destination d_0 also leads to channel A and is thus classified as a destination that may lead packets to a circular wait. The classification of channel C becomes $d_0 d_1 \mid d_1$. As there is a destination, namely d_0, that may lead packets to a circular wait, but for which there is no escape, channel C is marked deadlock-sensitive.

Deadlock-sensitive Channel c is classified with $x \mid y$ and $x \nsubseteq y$. There exists a destination that leads to a circular wait, but for which there is no escape.

Deadlock-immune Channel c is classified with $x \mid y$ and $x \subseteq y$. For all destinations, there is an escape.

After all marks have been determined, a deadlock can be constructed from all deadlock-sensitive channels. If all channels are deadlock-immune, the network is deadlock-free.

7.1.2 Example Trace

The algorithm consists of two steps. The first step is basically a depth-first search through the dependency graph. It expands a spanning tree, and after expanding the tree forwards, information is propagated backwards. The second step performs some post-processing. Initially, all channels are unmarked. Eventually, all channels get marked either deadlock-sensitive or -immune. We use a temporary mark "visited" for channels that have been expanded but whose marking is currently unknown. This ensures termination.

Consider the dependency graph in Figure 7.2a. Let the algorithm start in channel A. The different steps of the run of the algorithm are shown in Figure 7.3. The algorithm expands a tree spanning over the reach of A. It starts by marking channel A as visited (Step 1). Destinations d_0 and d_1 lead to channel B, and destination d_0 leads to channel E. To determine the classification of channel A, both neighbors must be expanded.

The algorithm proceeds with channels B and C and marks them as visited (Steps 2 and 3). From channel C, destination d_0 leads to E and destinations d_0 and d_1 lead to channel A. To determine the classification of channel C, both neighbors must be expanded.

First consider the expansion of channel D. As it is a sink, all destinations provide an escape. The classification of D becomes $\_ \mid d_1$ (Step 4). This classification produces two results (Step 5). First channel D can be marked as deadlock-immune, as the empty set is a subset of $\{d_1\}$. Secondly, this information can be propagated upwards in the tree towards channel C. All destinations leading from C to D lead to a deadlock-immune channel. Thus all these destinations (C, D) can be classified as leading from parent C to an escape. The classification of channel C becomes $\_ \mid d_1$.

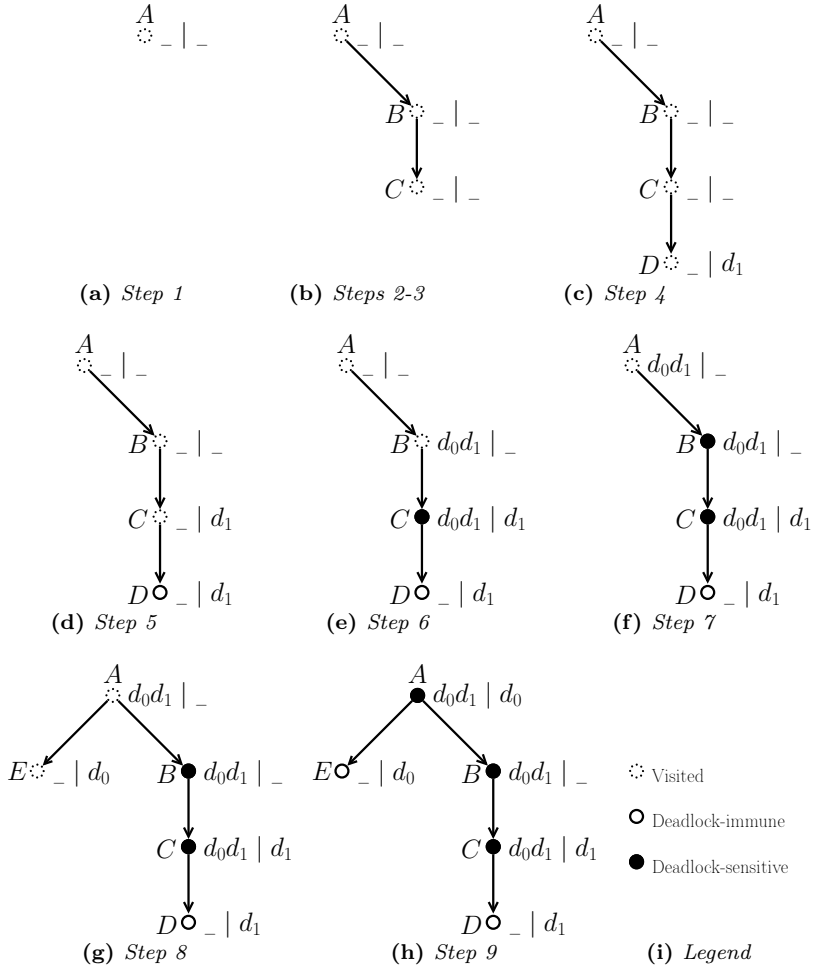


Figure 7.3: *Example trace. For sake of presentation, the set braces { and } have been omitted from the sets in the classifications. The underscore _ denotes the empty set to avoid similarity between ∅ and the vertices of the tree.*

Now consider the expansion of channel A. It has already been visited. This indicates a circular wait. Destinations d_0 and d_1, leading from channel C to channel A, are classified as such. The classification of parent C becomes $d_0 d_1 \mid d_1$. Again, two results are produced (Step 6). First, channel C is marked deadlock-sensitive as $\{d_0, d_1\} \not\subseteq \{d_1\}$. Secondly, since all destinations leading from B to C lead to a deadlock-sensitive channel, all these destinations are classified as leading into a circular wait. The classification of channel B becomes $d_0 d_1 \mid \_$.

As there are no other arcs going out of channel B, the mark of channel B is determined as deadlock-sensitive. This information is propagated upwards to channel A (Step 7). To complete the classification of channel A, channel E must be expanded as well (Step 8). As this is a sink, it is deadlock-immune. This information is propagated upwards to channel A, yielding the complete classification $d_0 d_1 \mid d_0$ of channel A. Channel A is marked deadlock-sensitive (Step 9).

As all channels have been marked, the algorithm terminates. It outputs a deadlock, consisting of the three deadlock-sensitive channels (see Figure 7.2b). In this deadlock, each deadlock-sensitive channel c with marking $x \mid y$ is filled with packets destined for a destination in x that is not in y.

7.1.3 Post-processing

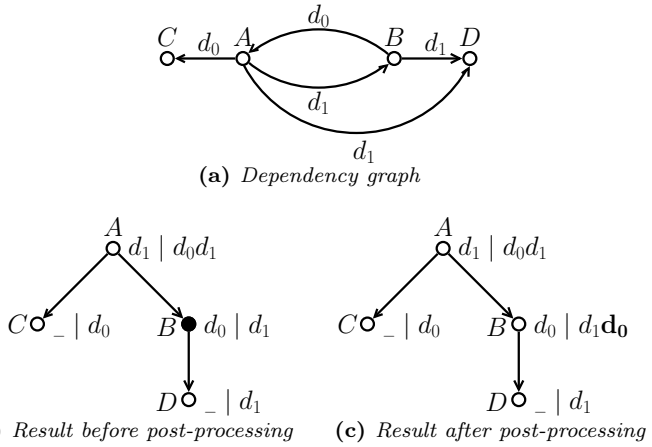
Consider the dependency graph in Figure 7.4a. We do not present the corresponding network and routing function, as it is just an artificial example used to show the need for a post-processing step. The network corresponding to this dependency graph is deadlock-free. To prove this, the only possible subgraph that needs to be considered is subgraph $\{A, B\}$. This graph has an escape, namely channel A. For all destinations, i.e., for both d_0 and d_1, there is a neighbor not in the subgraph.

We provide a trace where the algorithm yields an incorrect result without post-processing. The first step of the algorithm starts in channel A and marks it as visited. In this specific trace, the first neighbor of A that is expanded is channel B. Destination d_1 leads from channel B to a sink. Destination d_0 leads back to visited channel A. Consequently, the classification of channel B becomes $d_0 \mid d_1$ (see Figure 7.4b). Channel B is marked deadlock-sensitive, as $\{d_0\} \not\subseteq \{d_1\}$.

The algorithm continues with the expansion of the other neighbors of A. Since both destinations d_0 and d_1 lead to a sink, the classification of channel A becomes $d_1 \mid d_0 d_1$. Channel A is marked deadlock-immune. Channel B however, is still marked as deadlock-sensitive.

If the algorithm would stop here, it would conclude that this network is not deadlock-free as not all channels have been marked as deadlock-immune. The problem is that in this *specific* trace at the time channel B was marked deadlock-sensitive, channel A had not been expanded completely. In other words, it was not known at the time that channel A is deadlock-immune. A post-processing step is added to the algorithm to overcome this issue. This step adds to all deadlock-sensitive channels all destinations leading to deadlock-immune channels. In this step, a deadlock-sensitive channel can become deadlock-immune (see Figure 7.4c).

We claim that a network is deadlock-free if and only if after termination of the two steps of the algorithm all channels are marked deadlock-immune.



(a) *Dependency graph*

(b) *Result before post-processing* **(c)** *Result after post-processing*

Figure 7.4: *Example of a trace where post-processing is needed.*

7.2 Pseudo Code

Let us consider the first part of the algorithm, that is, the unfolding of the spanning tree (Algorithm 1). It takes two parameters: C_I is a set of channels that is to be explored. Arrays cyclics and escapes are assumed to be globally available. The algorithm keeps track of the parent of the channels in C_I with parameter p. This enables backwards propagation. The algorithm keeps expanding new neighbors until no unmarked neighbors exist. The current channel under investigation is c_0. For each channel, the algorithm stores the classification in two arrays. Array *cyclics* stores the destinations that lead into circular waits Array *escapes* stores the destinations that provide escapes. The marks of the channels are stored in array mark. There are four different marks:

0	**unmarked**	A channel is unmarked;
1	**visited**	A channel is visited, i.e, not all neighbors have been marked;
2	**immune**	All neighbors have been marked, the channel is deadlock-immune;
3	**sensitive**	All neighbors have been marked, the channel is deadlock-sensitive.

If channel c_0 is either a sink or deadlock-immune (Line 8), all destinations $\tau(p, c_0)$ provide escapes for p. Thus these destinations are added to escapes(p). Otherwise, channel c_0 is marked either **3** or **1** (Line 5). In the first case, c_0 has already been shown to be deadlock-sensitive. In the second case, it is unknown at this point whether c_0 is deadlock-sensitive or -immune. In both cases, all destinations $\tau(p, c_0)$ are considered to lead into circular waits from channel p. They are are added to cyclics(p). This consideration might be wrong. The post-processing step will check this and fix it if necessary.

If c_0 is neither a sink or marked, the algorithm continues its forwards expansion by expanding the neighbors of c_0 (Line 14). When this terminates, the gathered information is propagated backwards through the graph as follows: if there exists a

destination in cyclics(c_0) that is not in escapes(c_0), channel c_0 is deadlock-sensitive. Thus $\tau(p, c_0)$ is added to cyclics(p) and c_0 is marked with **3** (Lines 19–20). If the set of destinations cyclics(c_0) is a subset of escapes(c_0), channel c_0 is deadlock-immune. The destinations $\tau(p, c_0)$ is added to escapes(p). Channel c_0 is marked with **2** (Lines 16–17).

As for the post-processing step (Algorithm 2), this step initially considers all **3**-marked channels with **2**-marked neighbors. For all these channels it adds the destinations leading to **2**-marked neighbors, which have not been added already (Line 6). If, as a result of this, a channel c gets marked **2** (Lines 7–8), all parents of c have a new **2**-marked neighbor. Thus, all parents must be reconsidered (Line 9).

Main wraps up the two steps. It executes CreateTree for all unmarked channels c, with $C_I = A_{\mathtt{dep}}(c)$ and $p = c$. After this, it executes Post-Processing. It returns the – possibly empty – set of **3**-marked channels.

Algorithm 1 CreateTree(C_I, p)

Require: $C_I \subseteq A_{\mathtt{dep}}(p)$
1: **if** $C_I = \emptyset$ **then**
2: **return**
3: **else**
4: Pick element c_0 from C_I
5: **if** mark(c_0) $\in$ {**visited**, **sensitive**} **then**
6: cyclics(p) $\coloneqq$ cyclics(p) $\cup\, \tau(p, c_0)$
7: CreateTree($C_I - c_0, p$)
8: **else if** mark(c_0) = **immune** $\vee$ $A_{\mathtt{dep}}(c_0) = \emptyset$ **then**
9: mark(c_0) $\coloneqq$ **immune**
10: escapes(p) $\coloneqq$ escapes(p) $\cup\, \tau(p, c_0)$
11: CreateTree($C_I - c_0, p$)
12: **else**
13: mark(c_0) $\coloneqq$ **visited**
14: CreateTree($A_{\mathtt{dep}}(c_0), c_0$)
15: **if** cyclics(c_0) $\subseteq$ escapes(c_0) **then**
16: escapes(p) $\coloneqq$ escapes(p) $\cup\, \tau(p, c_0)$
17: mark(c_0) $\coloneqq$ **immune**
18: **else**
19: cyclics(p) $\coloneqq$ cyclics(p) $\cup\, \tau(p, c_0)$
20: mark(c_0) $\coloneqq$ **sensitive**
21: **end if**
22: CreateTree($C_I - c_0, p$)
23: **end if**
24: **end if**

7.3 Analysis

We prove that the computational complexity of our algorithm is O($|A|$), where A is the set of arcs in the dependency graph. We then prove correctness, by proving

Algorithm 2 Post-Processing(C_I)

```
1: if C_I = ∅ then
2:   return
3: else
4:   Pick element c_0 from C_I
5:   if mark(c_0) = sensitive then
6:     escapes(c_0) := escapes(c_0) ∪ {d ∈ τ(c_0, c_1) | mark(c_1) = immune}
7:     if cyclics(c_0) ⊆ escapes(c_0) then
8:       mark(c_0) := immune
9:       C_I := (C_I − c_0) ∪ {p ∈ parents(c_0) | mark(p) = sensitive}
10:    end if
11:    Post-Processing(C_I − c_0)
12:  else
13:    Post-Processing(C_I − c_0)
14:  end if
15: end if
```

Algorithm 3 Main

```
1: for all c_i ∈ C do
2:   if mark(c_i) = 0 then
3:     CreateTree(A_dep(c_i), c_i)
4:   end if
5: end for
6: Post-Processing(C)
7: return {c ∈ C | mark(c) = sensitive}
```

that Algorithm 3 returns the empty set if and only if the condition in Theorem 6.1 holds.

7.3.1 Computational Complexity

CreateTree visits each dependency arc exactly once, since after visitation a channel becomes permanently marked. The total running time of all calls of this step is therefore $O(|A|)$. It is basically a depth-first search, with backwards propagation.

The running time of the post-processing step is $O(|A|)$. A **32**-arc is an arc from a **3**-marked channel to a **2**-marked channel. The algorithm starts with **32**-arcs only. For all **32**-arcs, the algorithm adds the destinations labelling the arc to the escapes-array of the source of the arc (Line 6). The arcs considered in Line 6 are considered once: they could be permanently removed from the data-structure storing the graph. This holds since any **2**-mark is always stable. Line 9 adds new parents to C_I, thereby adding new arcs that are to be taken into consideration. All these arcs were initially **33**-arcs, but have just become **32**-arcs as channel c_0 has just been marked **2**. As the algorithm only considers **32**-arcs, none of these new arcs have been dealt with before. Each arc is considered at most once.

The running time of Main is the sum of the running times of CreateTree and Post-Processing. After post-processing it enumerates the **3**-marked channels in $O(|C|)$ time. Since the number of arcs is of a higher order than the number of channels, the total running time of the algorithm is $O(|A|)$.

Note that $O(|A|)$ is the order of the number of recursive calls. Each recursive call performs list operations such append and subset. However, no elements are deleted from a list. Using an advanced data structure such as a fibonacci heap, all the necessary list operations are $O(1)$ [56].

7.3.2 Correctness

The proof is structured in two parts: Lemma 7.3 states that our algorithm marks all channels **immune** if the network is deadlock-free. Lemma 7.4 states that any channel that can be part of a deadlock is marked **sensitive**. The proofs of these lemmas requires two auxiliary Lemmas 7.1 and 7.2.

The lemmas of the proof concern **2**- and **3**-marked channels. Mark **1** is only given during the execution of the algorithm, but will always be overwritten by either **2** or **3**. This is stated by our first lemma.

Lemma 7.1 After termination of CreateTree$(A_{\mathtt{dep}}(c_0), c_0)$ all channels in the reach of the channels in P are either marked **2** or **3**.

Proof. Any unmarked channel in the reach will eventually get marked **1**. Any **1**-marked channel will eventually become marked either **2** or **3**. A channel is only marked **1** on Line 13. Eventually the algorithm will reach either Line 17 or Line 20, where the channel is marked either **2** or **3**. Once a channel is marked **2** or **3**, it will never become either unmarked or marked **1**. □

We prove that if a channel c has escapes for all destinations in $\tau(c)$, i.e., if all destinations lead to a deadlock-immune neighbor, the channel will not be marked **3**. This lemma requires the post-processing step.

Lemma 7.2 After termination of Post-Processing, if for any channel c all destinations in $\tau(c)$ lead to a **2**-marked neighbor, channel c is not marked **3**.

Proof. The post-processing step ensures that for all **3**-marked channels the escapes array contains all destinations leading to **2**-marked neighbors. Since, by assumption, all destinations in $\tau(c)$ lead to a **2**-marked neighbor, all these destinations are included in escapes(c). Thus $\tau(c) \subseteq$ escapes(c). The algorithm classifies destinations only if they are in $\tau(c)$, i.e., it classifies a destination d only when it is actually possible for a message with destination d to reach channel c. Thus necessarily cyclics(c) $\subseteq \tau(c)$. By transitivity of $\subseteq$, we have established cyclics(p) $\subseteq$ escapes(p). As this is the criterion under which channels are not marked **3**, this implies channel c can never become marked **3**. □

We now prove that in a deadlock-free network, any channel gets marked **2**. Thus, the algorithm will return the empty set if the condition for deadlock freedom holds. Lemma 7.3 is proven as follows. If the algorithm marks a channel **3** and this marking is preserved by the post-processing step, it is possible to create a subgraph without an escape. This proof completely formalizes the intuition in Figure 7.2: a

deadlock is created from all deadlock-sensitive channels. Subgraph $S_{\mathbf{3}}$ is created by taking all **3**-marked channels. Lemma 7.2 proves that for each channel c in subgraph $S_{\mathbf{3}}$ there is a destination $d \in \text{cyclics}(c) \notin \text{escapes}(d)$ that leads only to **3**-marked neighbors. Since subgraph $S_{\mathbf{3}}$ contains all **3**-marked channels and since channel c has destination d which leads to **3**-marked channels only, channel c is not an escape for this subgraph. Since this holds for all channels c in subgraph $S_{\mathbf{3}}$, the subgraph has no escape.

Lemma 7.3 Assume all non-empty subgraphs have an escape. After termination of MAIN any channel c is marked **2**.

Proof. By Lemma 7.1, channel c is either marked **2** or **3**. The proof is by contradiction. Assume channel c is marked **3**. We prove that the set of **3**-marked channels does not have an escape. Let c' be any **3**-marked channel. By Lemma 7.2 there is at least one destination d that does not lead to any **2**-marked neighbor. By Lemma 7.1 destination d leads to **3**-marked channels only. Since there is a destination that does not lead outside of the subgraph consisting of all **3**-marked channels, channel c' is not an escape for this subgraph. This holds for all c' in the subgraph. Thus the subgraph does not have an escape. Furthermore this subgraph is not empty, since otherwise there would be no **3**-marked channels and channel c is marked **3**. Thus the assumption that all non-empty subgraphs have an escape has been contradicted. □

Lastly, we prove that if the algorithm returns the empty set, i.e., if all channels get marked **2**, the condition for deadlock-free routing holds. In other words, if the condition does not hold, there is some channel that will not be marked **2**. Lemma 7.4 states that any channel in a deadlock cannot be marked **2**. Since all channels eventually are marked either **2** or **3** (Lemma 7.1), any channel that can be in a deadlock gets marked **3**.

Lemma 7.4 If a channel c is in a subgraph S that has no escape, the channel will not be marked **2**.

Proof. The lemma holds initially since all channels are unmarked. We show by induction on CREATETREE that this lemma is preserved during this step. The exact similar argument holds for POST-PROCESSING. Thus the lemma is an invariant for the algorithm.

Assume that channel $c \in S$ and that S has no escape. The only reason a channel c gets marked **2** is when $\text{cyclics}(p) \subseteq \text{escapes}(p)$. We prove that this implies channel c is an escape. When channel c becomes marked **2** all neighbors of channel c have been explored. Thus all destinations in $\tau(c)$ are either in $\text{cyclics}(c)$ or in $\text{escapes}(c)$. Since by assumption $\text{cyclics}(c) \subseteq \text{escapes}(c)$, all destinations in $\tau(c)$ are in $\text{escapes}(p)$. If a destination is in $\text{escapes}(p)$ then it leads to a **2**-marked neighbor. Thus for all destinations, there is a **2**-marked neighbor. By the Induction Hypothesis, none of the channels in subgraph S are marked **2**. Thus for all reachable destinations there is a neighbor not in the subgraph: c is an escape for the subgraph. This contradicts the assumption that c is in a subgraph without an escape. Thus channel c cannot have been marked **2**. □

Together, Lemmas 7.3 and 7.4 state that a channel gets marked **3** if and only if it is in some subgraph without an escape. The algorithm returns the empty set if and only if all channels get marked **2**. By Theorem 6.1, the algorithm returns the empty set if and only if the network is deadlock-free. If a non-empty set is returned, a deadlock can be constructed from all **3**-marked channels.

Theorem 7.1 A network is deadlock-free if and only if MAIN returns the empty set.

$$\textsc{Main}(N) = \emptyset \iff \mathsf{DLF}(N)$$

Proof. Theorem 6.1 states that a communication network is deadlock-free if and only if all subgraphs have an escape. Assume all subgraphs have an escape. By Lemma 7.3 all channels will be marked **2** and thus MAIN returns the empty set. Assume MAIN returns the empty set. Then all channels are marked **2**. By Lemma 7.4 there is not a channel in a subgraph without an escape. Thus all subgraphs have an escape. □

We have defined a linear algorithm for deciding deadlock freedom of packet networks. We now proceed with wormhole networks.

Wormhole Switching

Detecting deadlocks in wormhole networks is much harder than detecting deadlocks in packet networks. The main cause of this increase in complexity is the requirement that in a legal wormhole configuration, worms do not intersect. In Section 7.7, we will show that this requirement makes deciding deadlock freedom in wormhole networks co-NP-complete. The search for an efficient polynomial algorithm can be stopped. If we want to define a notion of deadlock that is decidable in polynomial time, we have to drop this requirement. We call such a deadlock a *quasi-deadlock-configuration*.

The fact that worms may not intersect is directly expressed in Property (2) of Definition 6.17. It is also reflected in Property (1) which states that channel capacities may not be exceeded. This can be seen as follows. We prove our conditions and algorithms correct with parametric channel capacities, i.e., function $\text{cap} : C \mapsto \mathbb{N}^+$ is left generic. Thus, they also hold when the capacity of each channel is one. In a network where all channels have capacity one, requiring Property (1) is tantamount to requiring that all worms do not intersect.

Definition 7.1 A *quasi-deadlock-configuration* is a configuration which satisfies Properties (3) to (6) of Definition 6.17. A network is quasi-deadlock-free, notation $\mathsf{Q\text{-}DLF}(N)$, if and only if there exists no quasi-deadlock-configuration.

The necessary and sufficient condition presented in Theorem 6.2 is easily adjusted for quasi-deadlocks. We only have to lift the requirement that the set of routing paths is pairwise disjoint.

Theorem 7.2 A wormhole network is quasi-deadlock-free if and only if all sets of routing paths have an escape.

$$\mathsf{Q\text{-}DLF}(N) \iff \forall \Pi^\star \in \mathcal{P}(\mathcal{L}(C)) \cdot \left\{ \begin{array}{l} \Pi^\star \neq \emptyset \\ \mathbf{R}\text{-paths}(\Pi^\star) \end{array} \right. \implies \exists e \in C \cdot \text{esc}(e, \Pi^\star)$$

A quasi-deadlock is not necessarily an actual deadlock, as it might not be a legal configuration. However, if a network is quasi-deadlock-free, it is also deadlock-free.

$$\mathsf{Q\text{-}DLF}(N) \implies \mathsf{DLF}(N)$$

We present an algorithm for finding quasi-deadlock-configurations in wormhole networks. Using Theorem 7.2, it can prove a wormhole network deadlock-free in polynomial time.

7.4 Algorithm by Example

The wormhole algorithm is an extension of our algorithm for packet networks. Exactly as for the packet algorithm, for each channel c a classification $x \mid y$ is determined for any destination in $\tau(c)$. The crucial difference between the algorithms is that $x \subseteq y$ no longer guarantees that a channel can be marked deadlock-immune. If x is a subset of y, the set of destinations that lead messages into circular waits is a subset of the set of destinations for which there is an escape out of these cycles. Any *packet* can be routed towards an escape. However, in wormhole networks, a channel can be filled with tail flits. These flits cannot autonomously be routed towards an escape as they always follow the header flit. Therefore, a new mark *deadlock-attainable* is introduced. The intuition behind this mark is that in a channel no header flit can be permanently blocked, but it is possible that some tail flit is permanently blocked.

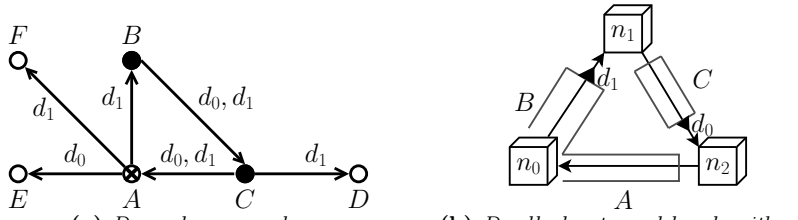
Let c be a channel with classification $x \mid y$, where x is a subset of y. No header flit can be permanently blocked. But if channel c is filled with tail flits, these can still be permanently blocked. Channel c is marked deadlock-attainable under the following two conditions. First, the tail flits must be part of some worm with its head in channel h. Secondly, the header flit in channel h must be permanently blocked.

The basic objective of our algorithm is to mark each channel as deadlock-immune, deadlock-sensitive, or deadlock-attainable. After termination of the algorithm, either all channels are marked as deadlock-immune and the network is deadlock-free or a deadlock can be created by filling all deadlock-sensitive channels with header flits and all deadlock-attainable channels with tail flits.

7.4.1 Deadlock-attainability

We recapitulate Example 6.3. Figure 7.5a shows the dependency graph of the network. Subgraph $\{A, B, C\}$ can form a deadlock consisting of two worms: $[B, A]$ and $[C]$. The first worm is destined for d_1, the second for d_0. Figure 7.6g shows the result of our algorithm. Channels B and C – the heads of the worms – have been marked deadlock-sensitive. Channel A – the tail of a worm – is marked deadlock-attainable. The other channels are deadlock-immune. Figure 7.5b shows the deadlock represented by these marks.

The algorithm knows that a header flit in channel h can be permanently blocked if channel h is marked deadlock-sensitive. Thus a channel is deadlock-attainable if there exists a routing path towards a deadlock-sensitive channel (see Definition 6.4



(a) *Dependency graph* **(b)** *Deadlock returned by algorithm*

Figure 7.5: *Example dependency graph. The black and crossed channels can form a deadlock.*

for a formal definition of routing path). This routing path can be established for some destination d, which is the destination of the worm. To know whether tail flits can be permanently blocked, there must be no escape for destination d at the *head* of the routing path.

Deadlock-sensitive Channel c is classified with $x \mid y$ and $x \nsubseteq y$. There exists a destination that leads into a circular wait, but for which there is no escape.

Deadlock-attainable Channel c is classified with $x \mid y$, with $x \subseteq y$, and there is a routing path for some destination d leading to a deadlock-sensitive channel h with classification $x' \mid y'$. Destination d must be in x', but cannot be a member of y'.

Deadlock-immune Channel c is classified with $x \mid y$, with $x \subseteq y$, and channel c is not deadlock-attainable.

After all marks have been determined, a deadlock can be constructed from all deadlock-sensitive and -attainable channels. In this deadlock, each deadlock-sensitive channel c with marking $x \mid y$ is filled with a header flit destined for a destination in x that is not in y. Each deadlock-attainable channel c is filled with tail flits belonging to a message with a header flit in some deadlock-sensitive channel. If all channels are deadlock-immune, the network is deadlock-free.

7.4.2 Example Trace

Figure 7.6 presents an example trace of our algorithm on the dependency graph in Figure 7.5a. The first six steps of the trace are equivalent to the example trace of the packet algorithm and will not be detailed (see Figure 7.3).

During Step 6, channel C has been marked deadlock-sensitive. Step 7 propagates this information upwards. As channel C is the only neighbor of channel B, the mark of channel B is determined as deadlock-sensitive. Destination d_1 is the only destination leading from A to B. The classification of channel A becomes $d_1 \mid \_$, indicating that destination d_1 leads from channel A into a circular wait.

To complete the classification of channel A, channels E and F must be expanded as well (Steps 8 and 9). As they are sinks, they are deadlock-immune. The classification of channel A becomes $d_1 \mid d_0 d_1$.

Channel A is marked deadlock-attainable. First, because $\{d_1\}$ is a subset of $\{d_0, d_1\}$. Secondly, because there is a routing path to deadlock-sensitive channel B.

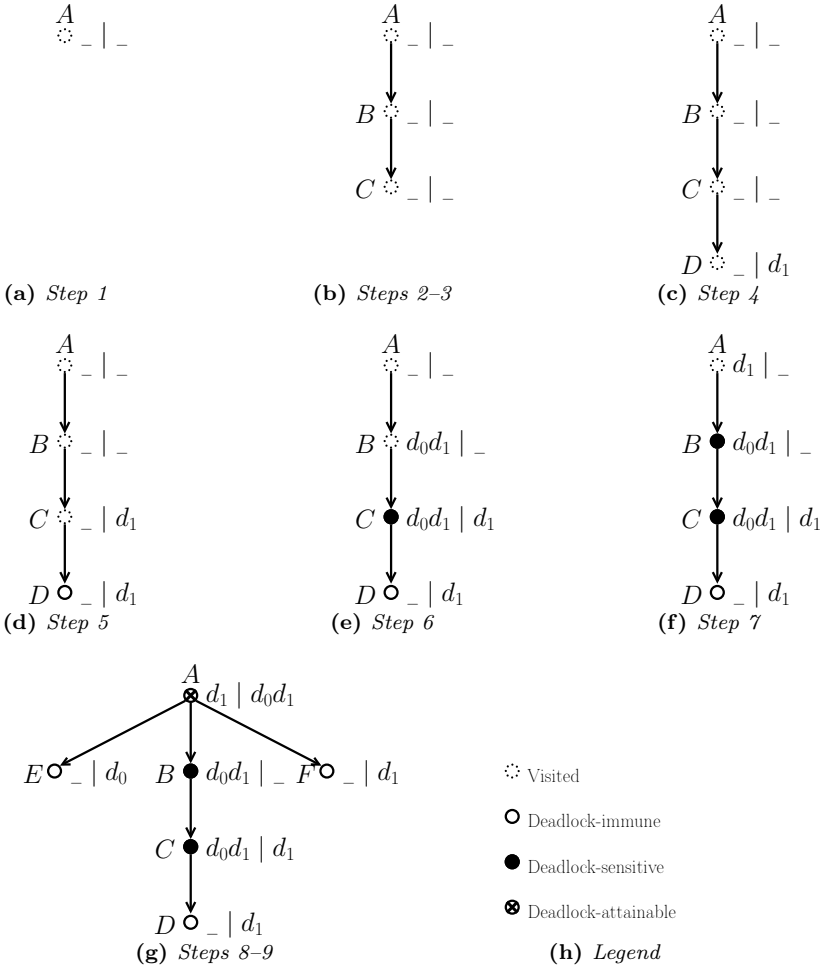


Figure 7.6: *Example trace. For sake of presentation, the set braces { and } have been omitted from the sets in the classifications. The underscore _ denotes the empty set to avoid similarity between ∅ and the vertices of the tree.*

This path can be established for destination d_1. Thirdly, because the classification of channel B is $d_0 d_1 \mid \_$ and destination d_1 is in $\{d_0, d_1\}$, and it is not in the empty set.

7.5 Pseudo Code

Algorithm 4 shows the first part of the algorithm. The algorithm uses the same data structures as the packet algorithm. A fourth marking is added.

4 **attainable** All neighbors have been marked, the channel is deadlock-attainable.

Algorithm 4 CREATETREE(C_I, pf)

Require: $C_I \subseteq A_{\mathtt{dep}}(p)$
1: **if** $C_I = \emptyset$ **then**
2: **return**
3: **else**
4: Pick element c_0 from C_I
5: **if** mark(c_0) $\in$ {**visited**, **sensitive**, **attainable**} **then**
6: cyclics(p) $\coloneqq$ cyclics(p) $\cup\, \tau(p, c_0)$
7: CREATETREE($C_I - c_0, p$)
8: **else if** mark(c_0) $= 2 \vee A_{\mathtt{dep}}(c_0) = \emptyset$ **then**
9: mark(c_0) = **immune**
10: escapes(p) $\coloneqq$ escapes(p) $\cup\, \tau(p, c_0)$
11: CREATETREE($C_I - c_0, p$)
12: **else**
13: mark(c_0) = **visited**
14: CREATETREE($A_{\mathtt{dep}}(c_0), c_0$)
15: **if** cyclics(c_0) $\not\subseteq$ escapes(c_0) **then**
16: cyclics(p) $\coloneqq$ cyclics(p) $\cup\, \tau(p, c_0)$
17: mark(c_0) = **sensitive**
18: **else if** cyclics(c_0) $\neq \emptyset$ **then**
19: cyclics(p) $\coloneqq$ cyclics(p) $\cup\, \tau(p, c_0)$
20: mark(c_0) = **attainable**
21: **else**
22: escapes(p) $\coloneqq$ escapes(p) $\cup\, \tau(p, c_0)$
23: mark(c_0) = **immune**
24: **end if**
25: CREATETREE($C_I - c_0, p$)
26: **end if**
27: **end if**

The difference between this algorithm and Algorithm 1 is in Lines 18 to 20. Line 14 continues the forward expansion by expanding the neighbor of channel c_0. When this terminates, the gathered information is propagated upwards in the tree as follows: if there exists a destination in cyclics(c_0) that is not in escapes(c_0),

channel c_0 is marked deadlock-sensitive. The set of destinations $\tau(p, c_0)$ is added to cyclics(p) and channel c_0 is marked **3** (Lines 16–17). This is still the same as for the packet algorithm.

The difference is that now, if there is at least one destination in cyclics(c_0) we give channel c_0 mark **4** (Lines 19–20). As there is one such destination, there is at least one neighbor that is not deadlock-immune. There is a path to a neighbor that is deadlock-sensitive. If the channel cannot be marked deadlock-attainable, it is marked **2** (Lines 22–23).

As for the post-processing step (Algorithm 5), this step initially considers all **3**- and **4**-marked channels with **2**-marked neighbors. For all **3**-marked channels it adds the destinations leading to **2**-marked neighbors to the escapes array (Line 6). If, as a result of this, the deadlock-sensitive destinations become a subset of the deadlock-immune destinations, the channel no longer must be marked **3**. The channel is marked **4**, if there exists a routing path from the current channel leading to a **3**-marked channel (Line 8). This path must adhere to the requirements for marking a channel deadlock-attainable: first, destination d for which the path is supplied must be a member of the cyclics array for the head of the path, but must not be an escape. Second, the path must at least be of length two. As the current channel is marked **3**, trivially there is a path of length one. However, the algorithm has just concluded that the **3**-mark is to be changed, thus this path is not valid. If there no such path, the channel is marked **2** (Line 12).

If the marking of a channel changes from **3** to **4**, all **4**-marked parents must be reconsidered (Line 10). Similarly, if the marking of a channel changes from **3** to **2**, all **4**- and **3**-marked parents must be reconsidered (Line 13). If the post-processing considers a **4**-marked channel, it checks whether there still exists a valid routing path to some **3**-marked channel (Line 17). If not, it is marked **2** and reconsiders all **3**-marked parents (Lines 18–19).

Main wraps up the two steps. It executes CreateTree for all unmarked channels c, with $C_I = A_{\mathsf{dep}}(c)$ and $p = c$. After this, it executes Post-Processing. It returns the – possibly empty – set of **3**- and **4**-marked channels.

7.6 Analysis

7.6.1 Computational Complexity

The algorithm consists of two steps. Algorithm 4, is basically a depth-first search with backwards propagation with O($|A|$) recursions.

The running time of the post-processing step is $O(|A||C|)$. The algorithm loops over a bag of arcs. It starts with **32**- and **42**-arcs only. For all **32**-arcs, the algorithm adds the destinations labelling the arc to the escapes-array of the source of the arc (Line 6). This can make the arc either a **42**-arc or a **22**-arc (Lines 9–12). For all **42**-arcs, the algorithm checks whether the parent of the arc is correctly labelled **4**. This can make the arc a **22**-arc. Arcs of type **22** have no need to be considered, as at all times a mark **2** is definite. Thus each **32**-arc is considered at most twice and each **42**-arc is considered at most once. This yields O($|A|$) recursions. The computational complexity of one recursive call is O($|C|$) in the worst-case, as in order to determine whether a channel must be marked

Algorithm 5 POST-PROCESSING(C_I)

1: **if** $C_I = \emptyset$ **then**
2: **return**
3: **else**
4: Pick element c_0 from C_I
5: **if** mark(c_0) = **sensitive then**
6: escapes(c_0) $\coloneqq$ escapes(c_0) $\cup$ $\{d \in \tau(c_0, c_1) \mid \text{mark}(c_1) = \textbf{immune}\}$
7: **if** cyclics(c_0) $\subseteq$ escapes(c_0) **then**
8: **if** $\exists \pi^d \cdot \begin{cases} \text{last}(\pi^d) = c_0 & \wedge \\ \text{mark}(\pi^d[0]) = \textbf{sensitive} & \wedge \\ d \in \text{cyclics}(\pi^d[0]) & \wedge \\ d \notin \text{escapes}(\pi^d[0]) & \wedge \\ |\pi^d| > 1 \end{cases}$ **then**
9: mark(c_0) = **attainable**
10: $C_I \coloneqq C_I \cup \{c \in \text{parents}(c_0) \mid \text{mark}(c) = \textbf{attainable}\}$
11: **else**
12: mark(c_0) = **immune**
13: $C_I \coloneqq C_I \cup \{c \in \text{parents}(c_0) \mid \text{mark}(c) \in \{\textbf{sensitive}, \textbf{attainable}\}\}$
14: **end if**
15: **end if**
16: **else if** mark(c_0) = **attainable then**
17: **if** $\neg\exists \pi^d \cdot \begin{cases} \text{last}(\pi^d) = c_0 & \wedge \\ \text{mark}(\pi^d[0]) = \textbf{sensitive} & \wedge \\ d \in \text{cyclics}(\pi^d[0]) & \wedge \\ d \notin \text{escapes}(\pi^d[0]) & \wedge \\ |\pi^d| > 1 \end{cases}$ **then**
18: mark(c_0) = **immune**
19: $C_I \coloneqq C_I \cup \{c \in \text{parents}(c_0) \mid \text{mark}(c) = \textbf{sensitive}\}$
20: **end if**
21: **end if**
22: POST-PROCESSING($C_I - c_0$)
23: **end if**

Algorithm 6 MAIN

1: **for all** $c_i \in C$ **do**
2: **if** mark(c_i) = 0 **then**
3: CREATETREE($A_{\texttt{dep}}(c_i), c_i$)
4: **end if**
5: **end for**
6: POST-PROCESSING(C)
7: **return** $\{c \in C \mid \text{mark}(c) \in \{\textbf{sensitive}, \textbf{attainable}\}\}$

deadlock-attainable the algorithm has to traverse the dependency graph once in search for a deadlock-sensitive channel.

The running time of Main is the sum of the running times of CreateTree and Post-Processing. After post-processing it enumerates all **3**- and **4**-marked channel in $O(|C|)$ time. The total running time of the algorithm is $O(|A||C|)$.

7.6.2 Correctness

Our algorithm is correct if it returns the empty set if the condition of Theorem 7.2 holds, and if a quasi-deadlock can be constructed from any non-empty result. First, we prove sufficiency, i.e., if the algorithm returns the empty set then the network is deadlock-free. This proof completely follows the intuition of Figure 7.5: after termination of the algorithm, a deadlock can be created from all deadlock-sensitive and deadlock-attainable channels. In this proof, we create a set of paths without an escape. We do not – and cannot – show that this set of paths is pairwise disjoint. Our algorithm returns true if and only if the network is *quasi*-deadlock-free.

Lemma 7.5 Assume all non-empty sets of routing paths $\Pi^\star$ have an escape. After termination of Main any channel c is marked **2**.

Proof. After termination of the Main, channel c is either marked **2**, **3** or **4**. The proof is by contradiction. Assume channel c is marked **3** or **4**. We prove that the set of **3**- and **4**-marked channels is a set of paths without an escape, contradicting the assumption. Take the set of paths $\Pi_{\mathbf{34}}$ obtained by taking for each **3**-marked channel c the singleton path $[c]$ and for each **4**-marked channel a routing path leading to a **3**-marked channel. The paths are chosen such that for each routing path $\pi \in \Pi_{\mathbf{34}}$ the destination for which the path can be established is in cyclics$(\pi[0])$ but not in escapes$(\pi[0])$.

Each **3**-marked channel c in the set of paths $\Pi_{\mathbf{34}}$ has a destination d that is a member of cyclics(c) and not a member of escapes(c), since channels are marked **3** only if cyclics$(c) \not\subseteq$ escapes(c). Since, if some destination leads to **2**-marked neighbors it is added to escapes(c), destination d does not lead to **2**-marked neighbors. Since destination d does not lead to **2**-marked neighbors, it leads to channels marked **3** or **4** only. Since 1.) the set of paths $\Pi_{\mathbf{34}}$ contains all **3**- and **4**-marked channels, since 2.) channel c is the head of a path, and since 3.) there exists a destination d such that all neighbors are included in $\Pi_{\mathbf{34}}$, we can conclude that channel c is not an escape for $\Pi_{\mathbf{34}}$.

For each **4**-marked channel c in the set of paths $\Pi_{\mathbf{34}}$, there exists a routing path π towards a **3** marked channel. This routing path can be established for some destination d. As destination d is a member of cyclics$(\pi[0])$ but not in escapes$(\pi[0])$, the head of this path has no escape.

Set of paths $\Pi_{\mathbf{34}}$ has no escape. It is non-empty, as by assumption channel c is marked **3** or **4**. This assumption is contradicted. Channel c is marked **2**. □

Secondly, we prove necessity, i.e., if our algorithm returns a non-empty set there exists a quasi-deadlock-configuration. We prove an invariant: if a channel is marked deadlock-immune, it cannot be a member of any set of paths without an escape. This invariant implies that if after termination of the algorithm all channels

are deadlock-immune, there cannot exist a non-empty set of paths without an escape. This implies deadlock freedom.

Lemma 7.6 If a channel c is in a set of paths $\Pi^\star$ that has no escape, the channel will not be marked **2**.

Proof. The lemma holds initially since all channels are unmarked. We show by induction on CreateTree that this lemma is preserved during this step. The exact similar argument holds for Post-Processing. Thus the lemma is an invariant for the algorithm.

Assume some set of routing paths Π, some channel $c \in \bigsqcup \Pi$, and assume that Π has no escape. The only reason channel c gets marked **2** is when 1.) $\text{cyclics}(c) \subseteq \text{escapes}(c)$ and 2.) there is no routing path towards a **3**-marked channel. We prove that these conditions imply there exists an escape for Π. As this contradicts our assumption, channel c cannot be marked **2**.

First, assume channel c is the head of one of the paths in Π. We show channel c is an escape. When channel c becomes marked **2** all neighbors of channel c have been explored. Thus all destinations in $\tau(c)$ are either in $\text{cyclics}(c)$ or in $\text{escapes}(c)$. Since by 1.) $\text{cyclics}(c) \subseteq \text{escapes}(c)$, all destinations in $\tau(c)$ are in $\text{escapes}(c)$. If a destination is in $\text{escapes}(c)$, it leads to a **2**-marked neighbor. Thus for all destinations, there is a **2**-marked neighbor. By the Induction Hypothesis, none of the channels in subgraph $\bigsqcup \Pi$ are marked **2**. Thus for all destinations there is a neighbor not in the subgraph. Channel c is an escape for Π.

Now assume channel c is *not* the head of one of the paths in Π. As channel c is in the tail of some path π, there exists a routing path from c to the head $h = \pi[0]$ of this path. By 2.) there exists no routing path towards a **3**-marked channel. Channel h cannot be marked **3**. We have already proven that channel h cannot be marked **2**, as channels at the head of one of the paths in Π cannot be marked **2**. We prove that it cannot be marked **4** either. For each **4**-marked channel, there exists a routing path towards a **3**-marked channel. Thus there exists a path from channel c to a **3**-marked channel x as well, by appending the path from c to h and the path from h to x. This contradicts 2.), which states there exists no routing path towards a **3**-marked channel. As channel h cannot be marked **2**, **3** or **4**, we have a contradiction. Channel c cannot be not at the head of one of the paths in Π

If channel c is in $\bigsqcup \Pi$, channel c is not marked **2**. □

These lemmas suffice to prove that the algorithm decides the necessary and sufficient condition of Theorem 7.2.

Theorem 7.3 A network is quasi-deadlock-free if and only if Main returns the empty set.

$$\textsc{Main}(N) = \emptyset \iff \mathsf{Q\text{-}DLF}(N)$$

Proof. Theorem 7.2 states that a communication network is quasi-deadlock-free if and only if all sets of routing paths have an escape. Assume all such sets have an escape. By Lemma 7.5 all channels will be marked **2** and thus Main returns true.

Assume MAIN returns true. Then all channels are marked **2**. By Lemma 7.6 there is no a channel in the union of a set of routing paths without an escape. Thus all sets of routing paths have an escape. □

As a direct consequence of Theorem 7.3, a network is deadlock-free if MAIN returns the empty set.

7.7 Proof of co-NP-completeness

To prove co-NP-completeness, we first define the complement of the decision problem, i.e., the problem of deciding whether there exists a deadlock in a wormhole network with adaptive routing. This problem will be referred to as WHS−DL. The proof is a standard problem reduction (see Figure 7.7). We reduce the set packing problem [83] to WHS−DL. That is, we assume there exists a machine $M_{\texttt{WHS-DL}}$ that can solve WHS−DL in polynomial time. We show that under this assumption the set packing problem can be solved in polynomial time as well, by building a machine $M_{\texttt{SP}}$. The set packing problem has been shown to be NP-complete [83]. The polynomial transformation from an instance of SP to an instance of WHS−DL is done by machine τ.

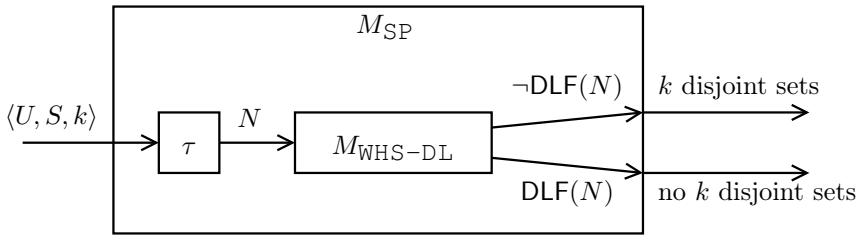


Figure 7.7: *A reduction of* SP *to* WHS−DL.

Definition 7.2 The *Wormhole Switching Deadlock Decision Problem* (WHS−DL) is defined as follows:

Given a communication network $N = (P, C)$ with routing function $\mathbf{R} : P \times P \mapsto \mathcal{P}(C)$, does there exist a deadlock configuration?

Definition 7.3 The *Set Packing Problem* (SP) is defined as follows:

Given a universal set $U = \{0, \ldots n\}$, a list of subsets $S = s_0 s_1 \ldots s_i$ where $s_j \subseteq U$ $(\forall_{0 \leq j \leq i})$, and an integer $0 < k \leq i$, does S contain k pairwise disjoint sets?

The idea behind the transformation is to create a network such that it contains deadlocks, but these deadlocks can only occur in one specific area in the network. This area contains a channel for each integer in U. We will refer to these channels as *U-channels*. The network is created in such a way that any deadlock requires exactly k worms. Furthermore, these worms have to cross the area containing the U-channels. The routing function creates a route through these channels for each

set in S. That is, for each set s_i the network contains a corresponding node s_i. This node is used as destination to route through the U-channels corresponding to set s_i. If there is a deadlock, there are k pairwise disjoint worms holding channels of paths created by the routing function for k destinations s_i, corresponding to k pairwise disjoint sets in S.

Transformation τ is first presented with an example. We give an instance of SP, i.e., a universal set U, a set of sets S, and a value for k, and transform this instance to an instance of WHS−DL, i.e., a network N with some routing function $\mathbf{R}$.

7.7.1 Transformation Example

Consider the following instance of SP: $U = \{0, 1, 2, 3, 4, 5\}$, $S = \{s_0, s_1, s_2\} = \{\{0, 1, 2\}, \{4, 5\}, \{2, 4\}\}$ and $k = 2$. There is a set of k pairwise disjoint sets, namely s_0 and s_1. We transform this instance to a network and routing function such that there is a deadlock configuration if and only if there are k pairwise disjoint sets in S.

Consider the network in Figure 7.8. Two nodes n_0 and n_1 are located at respectively the start and the end of an area consisting of five layers. The middle layer consists of a set of six U-channels corresponding to the six integers in U. Each set s_i has a corresponding node, also called s_i. As each set s_i has its own node – which is a destination for some messages – each set s_i can be associated with its own routing. More specifically, each set s_i has its own route to get from n_0 to n_1, through the U-channels of its set. For example, set $s_0 = \{0, 1, 2\}$ routes from n_0 to n_1 through U-channels 0, 1 and 2. At the second and fourth layer, messages destined for s_i are routed respectively towards the smallest element in set s_i, and from the greatest element in set s_i. The first and fifth layer ensure that there can be at most k worms traversing the area.

Furthermore, there is a node n_2, two channels A and B and a node x. These elements are needed to create a cycle in the network, thereby making deadlocks possible. Node x serves only as a destination for a worm holding channels A and B.

For sake of clarity, nodes x and s_i are not drawn in Figure 7.8. These nodes are used only as destinations and not part of any deadlock that might occur.

A worm is drawn in Figure 7.8 in thick lines. It is destined for node s_0. The corresponding set is $\{0, 1, 2\}$. At node n_0 it could have taken any of the two (k) channels going to layer 1. From the nodes at layer 1, only one channel can be taken by a message destined for s_0. This channel leads to layer 2, and then to the U-channel corresponding to integer 0, as 0 is the least element in s_0. Then a path from U-channel 0 to U-channel 1 to U-channel 2 is taken. After all U-channels corresponding to the set $\{0, 1, 2\}$ have been taken – in increasing order – the worm proceeds to any of the nodes at layer 4. Finally, its head is in the channel leading from layer 4 to node n_1.

Each worm that spans channels from n_0 to n_1 holds the U-channels corresponding to a set $s_i \in S$. The drawn worm holds U-channels 0, 1 and 2. A worm from n_0 to n_1 destined for s_2 holds U-channels 2 and 4. A legal configuration cannot have these two worms simultaneously, as both worms hold U-channel 2. Analogously, sets s_0 and s_2 are not pairwise disjoint.

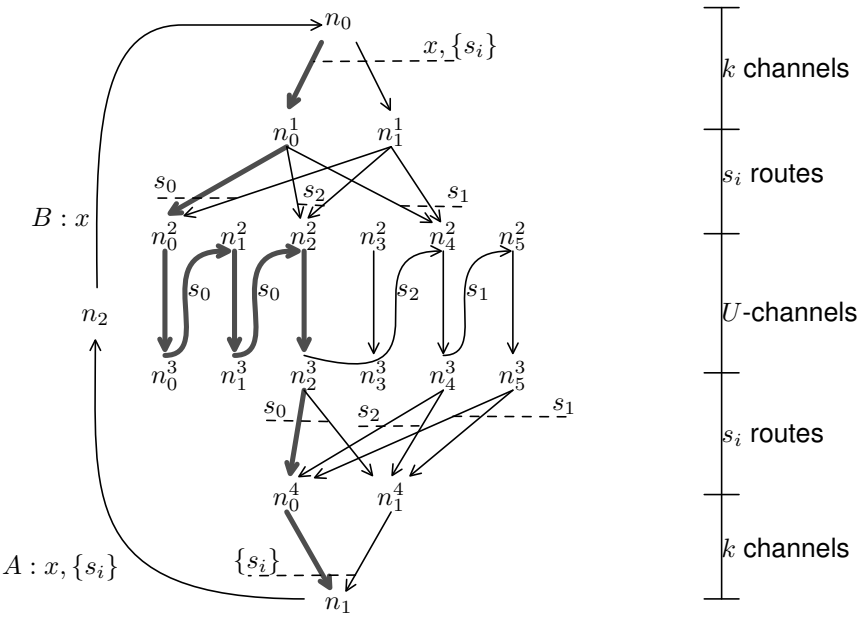


Figure 7.8: *Example of transformation τ*

Not all channels of the network are drawn. For all pairs of processing nodes (p_0, p_1) such that $p_0 \neq p_1$ and $p_0 \notin \{n_0, n_1\}$, there is a dedicated channel leading from p_0 to p_1. This dedicated channel is used only for messages destined for p_1. Thus, for all processing nodes other than n_0 and n_1 any message with any destination can be routed directly to its destination.

Dedicated channels cannot be involved in a deadlock. In a deadlock configuration all dedicated channels are empty, as otherwise a message arrives at its destination. A header of a message can only be permanently blocked in channels leading to either n_0 or n_1.

Consider the configuration where the two left channels A and B are filled with a worm destined for x. As n_0 has no outgoing dedicated channels, this worm needs to acquire one of the k channels going out of processing node n_0. If all these k channels are permanently blocked, a deadlock configuration is reached. However, if one of these channels contain a header, this header will be able to use a dedicated channel to arrive at its destination. Thus all these k channels must contain tail flits only. This holds for all channels leading from n_0 to the k channels going into n_1. In a deadlock configuration, all of these channels must contain tail flits only.

Besides channel B, the only channels that can contain a header flit that can be permanently blocked are the k channels going into n_1. As processing node n_1 has no outgoing dedicated channels, the headers in these channels must acquire channel A, which is held by a worm. Thus, any deadlock configuration in this network contains exactly $k + 1$ worms: one worm holding channels A and B and k worms holding channels from n_0 to n_1.

In this example, there were $k = 2$ pairwise disjoint sets in S, namely $s_0 = \{0, 1, 2\}$ and $s_1 = \{4, 5\}$. Thus the translated network must contain a deadlock

configuration. The channels of the worm concerning s_0 have been drawn in thick lines. There is also a worm for s_1 going through channels 4 and 5. These two worms, together with a worm holding channels A and B constitute a deadlock. The reason there is a deadlock is that the area of U-channels can be crossed by 2 worms that do not intersect.

In general, if there is a deadlock configuration, apparently there is a way to construct k worms crossing the area of the U-channels. These worms are pairwise disjoint. Each worm holds the channels belonging to one of the sets s_i. Thus k pairwise disjoint sets in S have been found. Assuming there is an algorithm that decides WHS−DL in polynomial time, it is possible to create an algorithm deciding SP in polynomial time. Problem WHS−DL is NP-complete. The complement of this problem, i.e., deciding deadlock freedom of adaptive routing functions in wormhole networks, is co-NP-complete.

7.7.2 Formal Proof

Theorem 7.4 WHS−DL is NP-complete. That is:

$$L_{\mathtt{SP}} \propto L_{\mathtt{WHS-DL}}$$

Proof. We define a polynomial transformation τ and prove that:

$$x \in L_{\mathtt{SP}} \iff \tau(x) \in L_{\mathtt{WHS-DL}}$$

This proof follows the exact intuition in Figure 7.8. First, we formally define the transformation τ. Secondly we show that, given k pairwise disjoint sets, we can create a deadlock configuration in the communication network. All six properties of Definition 6.17 hold for this configuration. Lastly, we show that any deadlock configuration necessarily has $k+1$ worms, where k worms cross the area of U-channels. Since these worms are – by Definition 6.17 – pairwise disjoint, k pairwise disjoint paths through the U-channels exist. Each path corresponds to a set in S. Thus there are k pairwise disjoint subsets in S.

Definition of τ

Given a universal set $U = \{0, \ldots, u'\}$, a set of subsets $S = \{s_0, \ldots s_{i'}\}$ and an integer k', a communication network N_τ is constructed: $N_\tau = (P_\tau, C_\tau)$.

We explain set of processing nodes P_τ by Figure 7.8. Nodes n_0 and n_1 are given. Node n_2 is the node between channels A and B. Nodes n_k^1 and n_k^4 $(0 \leq k < k')$ are the k' nodes at respectively the first and the fourth layer. Nodes n_u^2 and n_u^3 $(0 \leq u \leq u')$ are the $u'+1$ nodes at respectively the second and third layer. There is a node x, which serves as separate destination for the worm in channels A and B. Lastly, each node s_i $(0 \leq i \leq i')$ serves as destination for the worms crossing the area of U-channels.

$$\begin{aligned} P_\tau \quad = \quad & \{n_0, n_1, n_2\} \cup \\ & \{n_0^1, \ldots n_{k'-1}^1, n_0^2, \ldots, n_{u'}^2, n_0^3, \ldots, n_{u'}^3, n_0^4, \ldots, n_{k'-1}^4, x, s_0, \ldots, s_{i'}\} \end{aligned}$$

The set of channels C_τ is defined as follows. There is a channel from n_0 to each node n_k^1. There is a channel from each node n_k^1 to each node n_u^2. There is channel

for each pair (n_u^2, n_u^3). For all $u < v \leq u'$, there is a channel from node n_u^3 to n_v^2. There is a channel from each node n_u^3 to each node n_k^4. There is a channel from each node n_k^4 to n_1. Lastly, there are two channels A and B, respectively from n_1 to n_2 and from n_2 to n_0. All channels specified are identified by their processing nodes. E.g. channel A is identified as (n_2, n_0).

There is also a set of dedicated channels. For each set of processing nodes (p_0, p_1), where $p_0 \notin \{n_0, n_1\}$, there is a dedicated channel identified as $\mathcal{D}_{p_0 p_1}$. Nodes n_0 and n_1 have outgoing dedicated channels as well, but not leading to any of the nodes s_i or x. This concludes the definition of the communication network.

Routing function $\mathbf{R}_\tau$ is defined as follows: given a current node s and a destination d, always supply the dedicated channel $\mathcal{D}_{sd}$. Furthermore, adaptively supply the extra channels for the pairs (s, d) specified in Table 7.1.

(s, d)	Channels
$< n_0, s_i >$	$\{(n_0, n_k^1)\}$
$< n_0, x >$	$\{(n_0, n_k^1)\}$
$< n_k^1, s_i >$	(n_k^1, n_u^2) if and only if u is the least element of set s_i
$< n_u^2, s_i >$	(n_u^2, n_u^3) if and only if $u \in s_i$
$< n_u^3, s_i >$	(n_u^3, n_v^2) if and only if v is the next element in s_i after u
$< n_u^3, s_i >$	$\{(n_u^3, n_k^4)\}$ if and only if u is greatest element of s_i
$< n_k^4, s_i >$	(n_k^4, n_1)
$< n_1, s_i >$	A
$< n_1, x >$	A
$< n_2, x >$	B

Table 7.1

This routing function only requires the current node and the destination node to compute the set of next hops. Its type is therefore $P \times P \mapsto \mathcal{P}(C)$ and it satisfies our assumption that routing is memoryless. A worm spanning channels from n_0 to n_1 crosses the U-channels in increasing order.

Proof

($\Longrightarrow$)
Assume a universal set $U = \{0, \ldots, u'\}$, a set of subsets $S = \{s_0, \ldots s_{i'}\}$ and an integer k'. Also, assume that there exists k' pairwise disjoint sets in S. Let L be a list of k' indices such that for all indices i and j in L $(i \neq j)$, sets s_i and s_j have an empty intersection.

We show that the transformation has a deadlock. This deadlock is obtained by creating $k'+1$ worms. One worm, destined for destination x, holds channels A and B. Each worm w_k $(0 \leq k < k')$ is destined for $s_{L[k]}$. The tail holds the following channels: (n_0, n_k^1) and the intermediate path of (non-dedicated) channels supplied by the routing function from n_k^1 to n_k^4 for destination $s_{L[k]}$. The head of the worm holds channel (n_k^4, n_1). Each channel c of a worm is filled with exactly $\mathrm{cap}(c)$ flits.

To show that this is a deadlock, the six properties of a deadlock configuration have to be discharged. Properties (1) and (5) clearly hold. These respectively state that buffer capacities are not exceeded and that no header flit has arrived

at its destination. Property (2) states that each channel has flits belonging to one message only. Channels (n_0, n_k^1) and (n_k^4, n_1) are filled with flits belonging to worm w_k only. The channels of the intermediate paths are filled with flits belonging to one worm, as all sets $s_{L[k]}$ are pairwise disjoint. Property (3) holds, as for each worm w_k both the starting channel (n_0, n_k^1), the last channel (n_k^4, n_1) and the channels of the intermediate path are supplied for destination $s_{L[k]}$. The configuration is non-empty, as $k > 0$, and thus Property (4) holds. As for Property (6), the worm holding channels A and B has no available next hops as node n_0 has no dedicated channel for destination x and all channels (n_0, n_k^1) are filled with tails of the worms w_k. As for the worms w_k, as node n_1 has no dedicated channels for any of the destinations s_i, all these worms require channel A. This channel is unavailable. The created deadlock is a legal configuration which satisfies all six properties.

($\Longleftarrow$)

Assume a communication network N_τ and a routing function $\mathbf{R}_\tau$, result of the transformation $\tau(x)$, where $x = \langle U, S, k' \rangle$. Also, assume a deadlock configuration σ. We show that there exists k' pairwise disjoint subsets in S.

In N_τ, any deadlock configuration satisfies the following constraints. First, any deadlock configuration has k' worms, whose tails start in the channels (n_0, n_k^1), and whose heads are in the channels (n_k^4, n_1). Lastly, any deadlock configuration has a worm holding channels A and B.

This is shown by analysis of the possible locations of the headers of the worms in the deadlock. In a deadlock, a header cannot be in a dedicated channel, as this contradicts Property (5) of the deadlock definition. If the dedicated channel contains tail flits only, a header has arrived at is destination and all these flits will eventually evacuate the network. In a deadlock configuration all dedicated channels are empty.

This implies that any channel leading to a dedicated channel for all destinations cannot contain a header flit. Otherwise Property (6) is contradicted. Since all channels lead to dedicated channels for all destinations, except for channel B and the channels (n_k^4, n_1), the header flits of a deadlock must be in these channels.

There are no dependency cycles in the area of U-channels, as the routing function supplies all channels (n_u^2, n_u^3) in increasing order. Thus between nodes n_0 and n_1 there are no dependency cycles. The only other channels are A and B and the only other dependency is from A to B. Thus any cycle in the dependency graph of N_τ contains the dependency from A to B. As any deadlock necessarily contains a dependency cycle [36], channels A and B must be filled. Channel A cannot contain a header, implying that channels A and B contain a worm. This worm is destined for destination x, as channel B contains the header of this worm and can only contain messages destined for x. The worm cannot hold any other channels than A and B: all channels (n_k^4, n_1) are not supplied by $\mathbf{R}_\tau$ for destination x.

This worm has k' possible next hops: the channels (n_0, n_k^1). Thus all these channels must be filled with tail flits. This requires k' different worms, as it is impossible to route - for one destination - from channel (n_0, n_k^1) to another channel (n_0, n_l^1). This holds because channel B is only supplied for messages destined for x and channels (n_k^4, n_1) are only supplied for destinations s_i.

Thus σ has $k' + 1$ worms. Each worm w_k starts in node n_0 and ends at node

n_1. Each worm w_k has as destination s_i for some i, as its header is in some channel (n_k^4, n_1) and this channel can only be used by messages destined for some s_i. Furthermore, there are no two worms with the same destination s_i, as this implies they would intersect at the channels (n_u^2, n_u^3) with $u \in s_i$. Thus each worm w_k corresponds – one to one – to a destination s_i.

Each channel (n_u^2, n_u^3) corresponds to an element $u \in U$. Each destination s_i routes a worm through channel (n_u^2, n_u^3) if and only if $u \in s_i$. Since 1.) the k' worms are destined for different s_i, 2.) each worm holds a path from n_0 to n_1, 3.) each path from n_0 to n_1 for destination s_i holds channels corresponding to the set of s_i, and 4.) all worms are pairwise disjoint, there are k' sets in S that are pairwise disjoint. □

7.7.3 Deadlock Freedom versus Deadlock Prediction

The only other paper devoted to the computational complexity of deadlock related problems in wormhole networks is that of Di Ianni [78]. Di Ianni proves that *predicting* deadlock in wormhole networks is co-NP-complete. Arbib et al. prove a similar result for store-and-forward networks [2]. The deadlock prediction problem (DP) consists in deciding if a given configuration will necessarily result in a deadlock. Deadlock prediction could be of benefit to deadlock avoidance mechanisms, whereas deciding deadlock freedom of a routing function could be of benefit to deadlock prevention.

The result of this section – co-NP-completeness of deciding deadlock freedom – is not implied by co-NP-completeness of DP. For store-and-forward networks, DP is co-NP-complete whereas our problem can be solved in linear time by the algorithm presented in this chapter. The problems concern two related but inherently different questions. Problem DP concerns the question whether, given some configuration, all traces will eventually result in a deadlock configuration, i.e., whether some configuration is *bound* to deadlock. Our problem concerns the question whether there is deadlock configuration reachable from the empty configuration, i.e., whether the empty configuration *might* result in a deadlock.

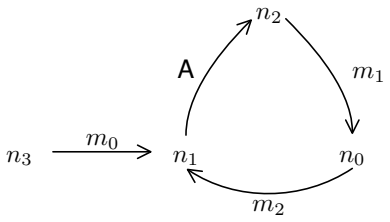


Figure 7.9: *Example configuration.*

We provide an example to stress the difference. Consider the network in Figure 7.9. The network contains three messages m_0, m_1 and m_2 destined for n_0, n_1 and n_2. Messages m_0 and m_2 arrive at intermediate node n_1. Both messages need to be forwarded to channel A. As this channel is empty, the configuration is not in deadlock yet. Depending on which message is chosen, the configuration results in a deadlock. If m_2 is chosen, it will arrive at its destination and eventually all

messages will evacuate the network. If m_0 is chosen, the cycle will be filled and all messages will be permanently blocked.

The deadlock prediction problem concerns the question whether this configuration is *bound* to end in a deadlock. In our example this is not the case, as there is a trace such that all messages evacuate the network. The deadlock prevention problem addressed in this section concerns the question whether there exists a reachable deadlock configuration. In our example this is the case. The two problems are inherently different from each other. A decision procedure for one of these problems cannot be applied to the other. The proofs are completely different as well.

7.8 Related Work

To the best of our knowledge, no deadlock detection algorithms exist for packet networks. For wormhole networks, the work of Taktak et al. is the only research related to automatic detection of deadlocks in a network model similar to our network model.

Taktak et al. first presented an algorithm for proving wormhole networks deadlock-free [137]. They extended Duato's condition (see Section 6.7) with dependencies between different types of messages. They assume that there exists an ordering between these message types [68]. Their condition is sufficient for deadlock freedom, but not necessary. An algorithm is presented that checks this condition automatically.

Taktak et al. continued with a polynomial algorithm checking a necessary and sufficient condition [138]. This condition – presented in Section 6.9 – does not include message dependencies. Their algorithm first breaks down the dependency graph into strongly connected components. A theorem is proven that it suffices to check for deadlocks in these strongly connected components instead of checking the dependency graph monolithically. They then search for a strongly connected component where there exists a channel that cannot be tagged according to their tagging condition.

As argued in Section 6.9, their condition is only sufficient, and not necessary. Their algorithm outputs false deadlocks. Indeed, their algorithm searches for quasi-deadlocks, just as the algorithm presented in this chapter.

The observable differences between the algorithms lie in the assumptions and the computational complexity. In contrast to our algorithm, Taktak et al. require the network to be livelock- and starvation-free. The computational complexity of their algorithm is $O(|P|^4)$, where P is the set of processing nodes in the network. Our wormhole algorithm is one degree lower in complexity. Taktak et al. implemented their algorithm in the tool ODI and provide experimental results. In the next chapter, we present industrial and real-life applications of our algorithms. Experimental results will be given, and we will compare the performance of our algorithms with that of Taktak et al.

7.9 Conclusion

We have presented algorithms to search for deadlocks in communication networks. For packet networks, we defined an algorithm with a computational complexity linear in the number of channels in the network. For wormhole networks, we have shown that a polynomial algorithm does not exist, assuming co-NP does not equal P. This is due to the fact that worms cannot intersect. We have defined the notion of quasi-deadlocks, where this fact is disregarded. A polynomial algorithm has been defined to search for quasi-deadlocks in wormhole networks.

Given a specification of the topology and the routing function, our algorithms are push-button solutions for finding deadlocks. In case of deadlock, our algorithms provide detailed feedback. These properties, and their low computational complexity, makes them usable as debugging tools for designers.

At this point, two versions of both algorithms exist: an efficient implementation in C and a non-executable specification in ACL2. Our utmost objective is to make the ACL2-specification efficiently runnable. If the ACL2-implementations decide deadlock freedom, this result can be regarded as a mechanized proof. This way, we can have formal verification play a useful role in the design process of Networks-on-Chip.

The next chapter will present some applications of these algorithms. An interesting application of our fast algorithms is the verification of fault-tolerant network designs or designs with irregular routing.

In Part II, productivity is broken down into several smaller proof obligations. The largest and most difficult proof obligation to discharge is deadlock freedom. The algorithms in this chapter solve this issue. In the next chapter, we present the tool DCI2, which uses the algorithms presented in this chapter to discharge all proof obligations required for proving productivity. This yields a push-button solution for proving productivity of communication networks.

Chapter 8
Applications

The previous chapter presented deadlock detection algorithms. The purpose of these algorithms is to be of use to a designer of communication networks. In this chapter, we incorporate the algorithms in the tool DCI2. DCI2 adds various features to the algorithms that are necessary to make them valuable in practice.

We provide experimental results on some basic benchmarks on six different routing functions in four different topologies. We then present two more complex examples in detail. Both examples are based on the Network-based Processor Array (NePA) [4, 3, 89]. The first example extends the traditional NePA architecture with a routing function that dynamically routes around faults. Even though this introduces circular dependencies in some faulty configurations, the network remains deadlock-free. This routing logic is the result of a collaboration with the University of California, Irvine (UCI). The second example – taken from Wang et al. [144] – extends NePA with on-chip wireless communication. Because not all processing nodes have wireless routers, routing becomes irregular and difficult to verify manually. DCI2 efficiently finds deadlocks.

8.1 DCI2

A major extension of our deadlock detection algorithms deals with the possibility to verify fault-tolerant routing functions. The design of a deadlock-free fault-tolerant routing function is often difficult, as absence of deadlocks must be assessed for many different faulty configurations. The design might be deadlock-free in many cases, but a specific combination of faults may cause a deadlock. DCI2 can aid in both the design and the verification of such routing functions.

DCI2 takes as input a description of the topology and a C^{++} implementation of the routing function. Our tool has the following features:

- The tool checks whether the model of the routing function is correct with respect to the topology. In the hardware it is impossible for a packet to be routed through channels that do not exist. The model written in C^{++} code however can route, e.g., a packet west at the left-most column, or route diagonally even when a diagonal channel does not exist. The tool detects such errors and outputs them.

- The tool takes as input a basic *fault model*. Common fault models are "assume at most two channels can be faulty" or "assume at most one router is faulty". The tool checks the number of available cores in the current machine, splits up the set of faulty configurations to check under the fault model, and executes the algorithm on these configurations *in parallel*. It provides 100% coverage in the cases considered.

- The tool checks whether the routing function is *connected*. In a fault-tolerant design, it is possible that in some combination of faults, a packet arriving at a processing node has no path to its destination. If this is the case, the tool outputs exactly which packet in which channel with which destination has no next hops.

- The tool detects *livelocks*. In fault-tolerant designs, livelock can be hard to find manually. When encountering faults, packets may need to be sent back to where they came from. During the computation of the dependencies, the tool checks whether scenarios exist in which one packet gets into a cycle. For the isolated network model, this is a necessary and sufficient condition for livelock. If a livelock has been found, we report the exact trace leading to this cycle and the cycle itself.

- The tool provides quantitative output. In fault-tolerant designs, one may be interested to know how many of the faulty configurations yield a deadlock or livelock. We have added a command-line option to let the algorithm assemble quantitative information on how many of the configurations are correct, how many contain deadlocks, livelocks, etc. In the next section we will provide more details on this.

Because DCI2 requires only little user input and it returns results quickly, it can be used as a debugging tool during the design of interconnects and routing functions. Our fault-tolerant routing function has been obtained this way: start with a simple routing function, and then recursively detect erroneous scenarios and adapt the routing logic until a correct result has been obtained.

8.2 Benchmarks

Figure 8.1 shows results on some standard routing functions. All experiments have been performed on a 2.93 GHz Intel Core 2 Duo computer, with 2 GB memory. We present the results on the wormhole algorithm. For the packet algorithm, all running times are less or equal.

2D Mesh – XY [103]
XY routing is a deterministic routing function for two-dimensional meshes used in, e.g., the HERMES chip [99]. Messages are routed first along the X-axis and then along the Y-axis. It is deadlock-free, as there are no cyclic dependencies. We have verified this for a mesh with 4225 nodes and 16900 channels in 0.31 seconds.

2D Mesh – West-First [61]
West-first routing is an adaptive routing function for two-dimensional meshes. Our algorithm needed 1.49 seconds for a 65 by 65 mesh with 16900 channels.

2D Mesh – Shortest Path [138]
Shortest path routing is an adaptive routing function, which routes messages along the shortest path. It is not deadlock-free. Our algorithm finds a deadlock in 0.87 seconds for a 65 by 65 mesh with 16900 channels.
Double 2D Mesh – Shortest path with XY [138]
There are two virtual channels in each direction between the processing nodes. One of the channels is used for the adaptive shortest path routing function. The other channel is used for the deterministic XY routing function. The network is deadlock-free, even though there are cyclic dependencies. Our algorithm returns true in 110,07 seconds for a 45 by 45 mesh with 16200 channels.
Spidergon – Shortest path [31, 32]
Spidergon STNoC is an architecture developed by STMicroelectronics. The Spidergon topology is a ring where each node has a channel going clockwise, counter-clockwise, and across. Its shortest path routing function is deterministic. We have formalized an implementation of the Spidergon without virtual channels. This implementation has deadlocks, and we checked this for the Spidergon chip with eight processing nodes, i.e., the Octagon chip [82].
Double ring – AAD
We have designed a new adaptive routing function for Spidergon. There are two virtual channels in the counter- and clockwise-direction between the processing nodes on the ring. AAD routing – for Adaptive-Across-Deterministic – routes a message first adaptively in any direction. Once it has taken an across-channel, it is deterministically routed towards its destination. Our algorithm proves that it is deadlock-free in 65,12 seconds for a network with 2048 nodes and 12288 channels.

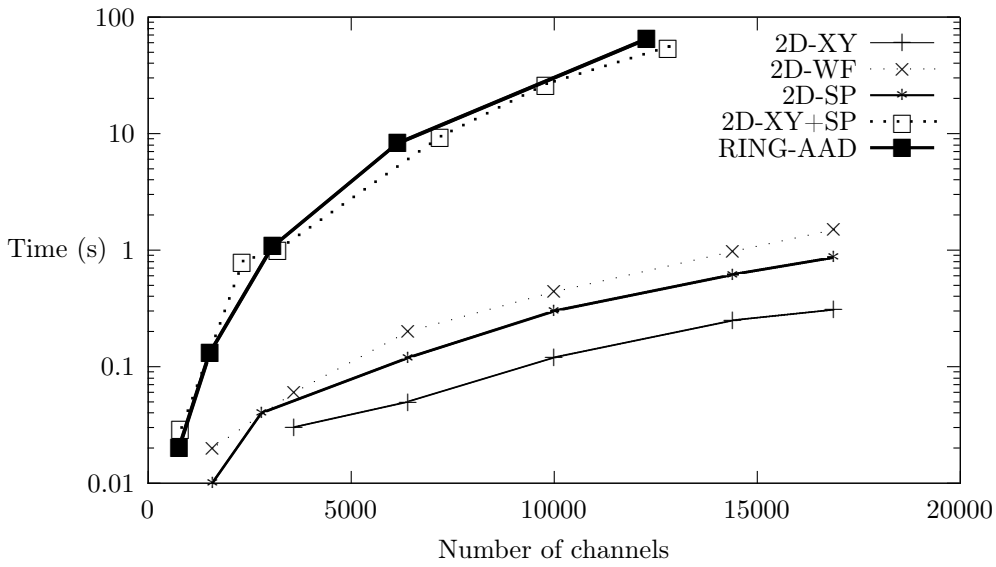


Figure 8.1: *Experimental results*

As Figure 8.1 shows, our algorithm performs significantly better on XY and

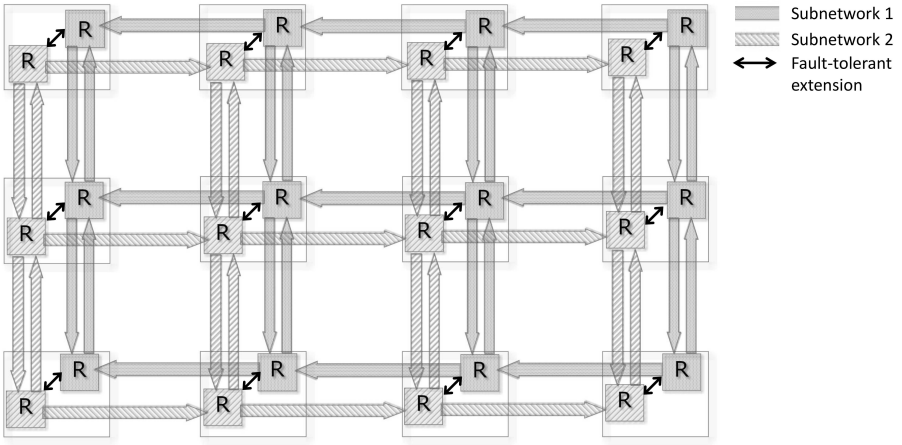


Figure 8.2: *NePA Mesh Network, where the bold arrows denote physical links between the subnetworks used for fault-tolerant routing.*

West-First routing than on "Shortest Path with XY" and AAD routing. This is due to the fact that the first two routing functions are deadlock-free because there are no cyclic dependencies. In such cases, our algorithm performs exactly like a regular cycle detection algorithm and terminates in linear time.

8.3 NePA with Fault-tolerant Routing

We shortly introduce the NePA architecture. Both this architecture and the extensions presented in this chapter are realistic and feasible examples of Networks-on-Chip, as they have been implemented in synthesizable HDL. For a discussion on feasibility, performance and scalability, we refer to the original paper of Bahn et al. [4].

In the NePA NoC system, processing nodes are mapped to a two dimensional mesh network of routers. The routers are connected through physical links, i.e., the channels. Horizontally, there is one eastern and one western channel between two adjacent routers. Vertically, there are two northern and two southern channels.

The NePA NoC design can be viewed as two subnetworks where Subnetwork 1 transports eastbound traffic and Subnetwork 2 transports westbound traffic. Figure 8.2 illustrates connections between routers. As the two subnetworks separate east- and westbound traffic, no dependency cycles occur. Consequently there are no deadlocks. The fault-tolerant extension presented in this section introduces physical links to add the capability to switch between subnetworks. These new links introduce dependency cycles.

Figure 8.3 shows the design of a router. Each router has ports for each incoming channel. For example, the western in-port contains traffic that is going in the eastern direction. The router is divided into two subrouters, one for each subnetwork. For example, as the western in-port contains traffic heading in the

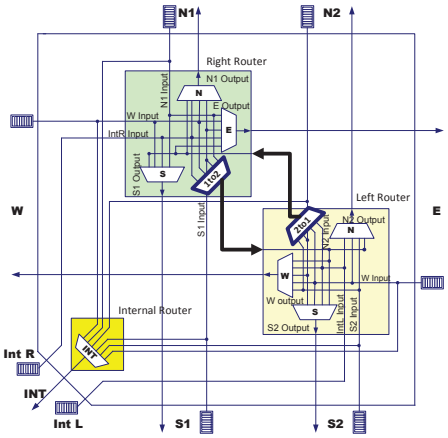


Figure 8.3: *NePA Router, where the bold arrows denote physical links between the subnetworks used for fault-tolerant routing [4].*

eastern direction, it is connected to the subrouter belonging to Subnetwork 1.

In a subrouter, each in-port is connected to a Header Processing Unit (HPU) which can receive a header flit, process the destination address and send out a request to the proper output ports. There is one HPU per incoming channel. The arbiters of the output ports grant one of the requests from the HPUs and reserve the output port until a packet transfer is completed. After that, the output port is released for other packets.

Figure 8.4 shows a detailed block diagram of a subrouter. Highlighted is the HPU that is connected to the the western in-port. The east-bound traffic in this port is directed to the north, east, or south arbiters of Subnetwork 1. These requests are denoted by *W_to_N1_req*, *W_to_E_req*, and *W_to_S1_req*. Traffic can also be directed to the local arbiter connected to the processing node.

To support fault-tolerant routing, some modifications of the original design are needed. Information on faulty links is passed between routers through wires. We assume that the NoC architecture is equipped with Design For Test technology [142, 66]. These Built-In-Self-Test (BIST) mechanisms detect faults in links and switches and update the links' status information accordingly. Between two subrouters, physical links are added to move a packet between the two subnetworks (see Figure 8.3). A fifth outgoing link **W_to_SUB2_req** is added to the HPU depicted in Figure 8.4 to request a switch from Subnetwork 1 to Subnetwork 2.

8.3.1 Routing Logic

The routing logic is contained in the HPU structures. Figure 8.4 shows the HPU that is connected to the western in-port. We show the details of the routing logic for this HPU, other HPUs are similar.

The intuition behind our routing logic is the following:

1. If there are no faulty channels, always supply all minimal routes.

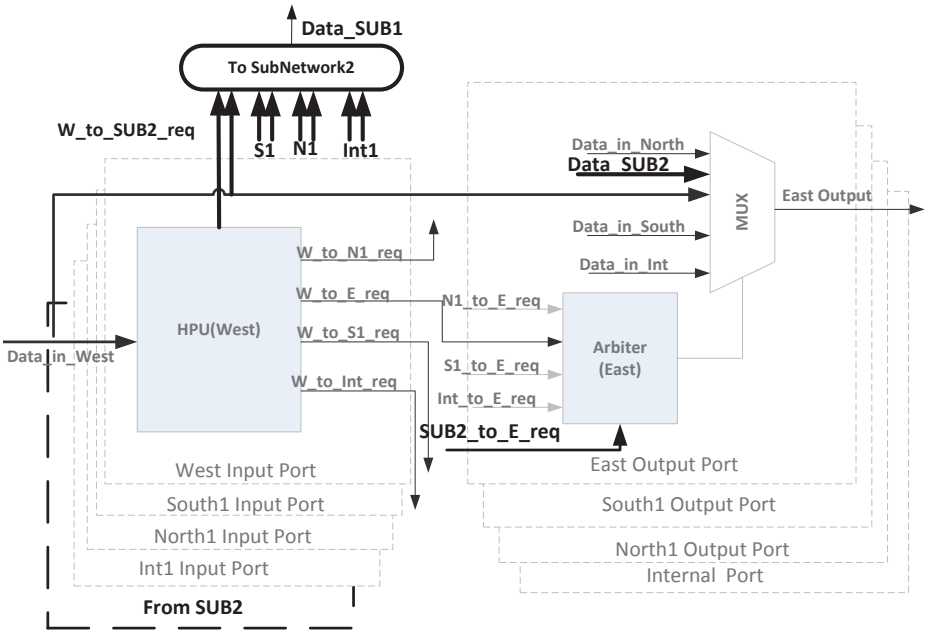


Figure 8.4: *Header Processing Unit, where the bold links are used to switch to the other subnetwork for fault-tolerant routing.*

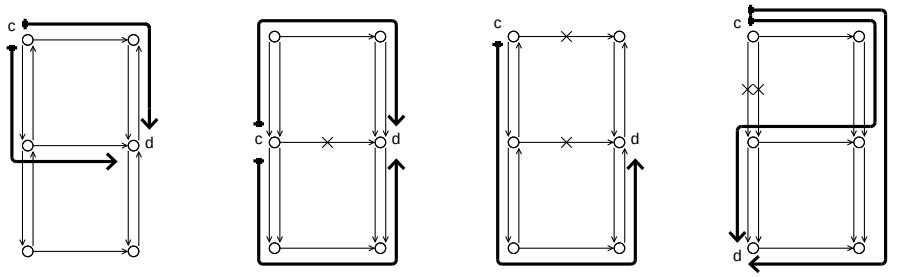
2. If there are faulty channels, take all shortest routes around these faulty channels that do not cause deadlock or livelock.

Routes are restricted only when they are the cause of deadlock or livelock. Since many different faulty configurations exist, yielding many exotic corner cases, the routing logic becomes very intricate and extensive. It has been obtained by continuously debugging the current routing logic, adding case distinctions where necessary, and rerunning DCI2.

If there are no faults, the routing logic equals a minimal adaptive shortest path routing logic. This routing logic can already handle some faults, as there may be alternative paths to reach a destination. In some cases, the minimal shortest path routing does not provide more than one path. For example, when the source and destination are in the same row or column, one fault may cause the routing function to become disconnected. Two approaches can make the routing logic fault-tolerant: either more paths must be supplied or the routing logic must prevent routing towards an area where the destination is no longer reachable.

Starting with the minimal routing logic, we have incremented the logic step-by-step with case distinctions to handle faults. Each adaptation is designed in such a way that it takes minimal routes around the set of faults. Figure 8.5 shows routes supplied by the routing logic in different faulty configurations. With no faults, all minimal routes are supplied (see Figure 8.5a). In Figure 8.5b there is a faulty channel between the current position and the destination. When multiple routes around a fault are possible, all routes that do not cause deadlocks or livelocks will

be supplied. Figure 8.5c provides an example where there are two faults between the current position and the destination. In this particular case, the length of the minimal route increases due to the faults. Figure 8.5d provides an example where switching between subnetworks is inevitable. Consequently, dependency cycles are inevitable as well.



(a) *No faults* **(b)** *One horizontal fault* **(c)** *Two horizontal faults* **(d)** *Two vertical faults*

Figure 8.5: *Routes supplied by the routing logic. Processing nodes c and d are the current node and the destination node respectively. The crossed channels are faulty.*

The eastbound routing logic is split up into three parts: the destination is either south-east of the current processing node, north-east, or east at the current row. We start with describing the south-east logic (see Algorithm 7).

Signal OK(d) returns 1 if and only if the channel indicated by direction d exists and is non-faulty. If a second direction is supplied, OK(d_1, d_2) returns 1 if and only if the channel reached by first performing a step in direction d_1 and then in direction d_2 exists and is non-faulty. For example, at router $(0, 1)$ signal OK($N1$) will return 1 if and only if the north channel of Subnetwork 1 from $(0, 1)$ to $(0, 0)$ is non-faulty. Signal OK($N1, N1$) will return 0, as $(0, 0)$ has no northern channels. A request to an arbiter is performed by setting the corresponding request bit to 1, i.e., the statement "$W\_to\_S1\_req = 1$" requests the southern channel of Subnetwork 1 of the current router.

Without any faults, we can simply supply the eastern and the southern channels. Line 2 supplies the eastern channel as next hop. The conjunction in Line 1 contains the conditions under which we route east. First of all, the eastern channel must exist and be non-faulty, i.e., OK(E) must return 1. The second conjunct prevents routing eastwards in one particular case. Consider Figure 8.6a. The destination is in the column at the end of the eastern channel and the two channels leading south are both faulty. If we route east, then subsequently we have to route around the two faulty channels. This rerouting means that either we go east or west. Going east results in three extra hops and one extra switch from Subnetwork 2 to Subnetwork 1. It also yields a livelock. Going west results in three extra hops and two switches between subnetworks. We therefore prevent going east, if the destination is in the next column and the two southern channels are faulty.

A similar restriction is needed for turning south (Line 4). This restriction is more severe: We never route south, if the channel in eastern direction at the end of the southern channel is faulty. This is to exclude some livelocks. An example

Algorithm 7 South-East Routing Logic of West In-Port

```
1: if OK(E)∧
   (dx > cx + 1 ∨ (dx = cx + 1 ∧ (OK(E, S1) ∨ OK(E, S2)))) then
2:    W_to_E_req = 1;
3: end if
4: if OK(S) ∧ OK(S, E) ∧ (! OK(E) ∨ OK(S2)) then
5:    W_to_S1_req = 1;
6: end if
7: if W_to_E_req = 0 ∧ W_to_S1_req = 0 then
8:    if OK(N1) then
9:       W_to_N1_req = 1;
10:   else
11:      Set intermediate destination to south
12:      W_to_SUB2_req = 1;
13:   end if
14: end if
```

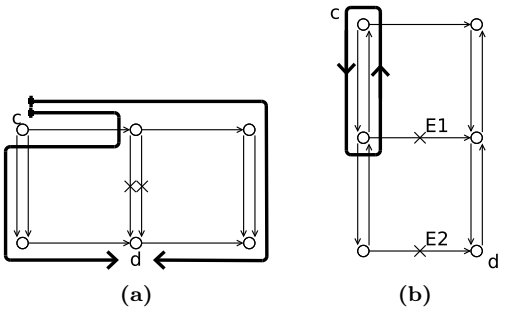
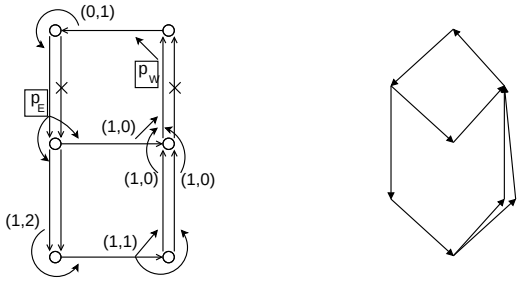


Figure 8.6: *Routes* prevented *by the routing logic. Processing nodes c and d are the current node and the destination node respectively. The crossed channels are faulty.*

occurs if – at the bottom of the mesh – both the eastern channels of the bottom two rows are faulty (see Figure 8.6b). If we would route south from node c even though channel $E1$ is faulty, we end up in a situation where the only possible next route is a 180 turn back north leading back to node c. As we do not remember the path taken by the packet, the routing logic will supply the south channel again, yielding a livelock.

Line 4 enforces a second restriction for going south. A packet is not routed towards the southern direction if one of the southern channels is faulty, when the eastern route is available. This restriction prevents some peculiar deadlocks which occur with some specific combinations of two faulty channels. Consider Figure 8.7a, where channels $(0, 0, S2)$ and $(1, 1, N1)$ are faulty. Let p_E denote an eastbound packet occupying channel $(0, 0, S1)$. Without the restriction, packet p_E can be routed towards both the eastern and the southern directions. In this particular deadlock configuration, it has chosen to go south and occupies $(0, 0, S1)$. As there is only one available southern channel from $(0, 0)$, all western going traffic uses this channel as well. So an eastbound packet in $(0, 0, S1)$ blocks westbound

traffic. Similarly, all eastbound traffic now uses channel $(1, 1, N2)$ since channel $(1, 1, N1)$ is faulty. So a westbound packet in $(1, 1, N1)$ can block eastbound traffic. We will refer to this packet with p_W. This completes a circular wait. However, for adaptive routing, a circular wait is not sufficient for a deadlock. The minimum deadlock in this particular configuration consists of three intertwined circular waits.



(a) *Deadlock with two faulty channels* **(b)** *Channel dependency graph*

Figure 8.7: *Deadlock occurring with two faults and the dependency graph depicting the three circular waits that constitute the deadlock.*

Due to these restrictions, in some cases no next hops have been supplied at this point (Line 7). For example, this is the case when both the eastern channel and the southern channel in Subnetwork 1 are faulty. To minimize the number of subnetwork switches, we first try to route north in the current subnetwork (Line 9). If this channel is faulty or if we are at the top of the mesh, we switch subnetwork and route south (Lines 11 and 12).

Algorithm 8 provides the routing logic in case the destination is eastbound on the current row.

If the eastern channel is not faulty, we simply go east (Line 2 of Algorithm 8). If this is not possible, then it must be determined whether the packet is routed north or south around the faulty channel. To prevent livelocks, the next hop is chosen such that is has at least one non-faulty channel in the current subnetwork (Lines 4 and 7). As shown in Figure 8.8, if a packet destined for $(1, 1)$ is injected in $(0, 0)$ it is first routed south as the destination is south. At $(0, 1)$, the packet must be prevented from going back north, as otherwise a livelock occurs. As in this case both $\mathrm{OK}(N, E)$ and $\mathrm{OK}(N, N1)$ will yield false, the packet will not be routed north.

This will always route around a faulty eastern channel, except when the packet is at either the bottom or the top of the mesh (Lines 11 and 14). Consider the case at the top of the mesh. Channel S1 is faulty as otherwise Line 7 would have applied. So a switch between subnetworks is necessary. The routing logic requests channel S2. Similarly, at the bottom of the mesh, channel N2 is requested.

At this point, the south-east and the east routing logic have been described. The north-east logic is similar to the south-east logic. Two cases are left: the destination is north or south in the current column. We describe the southern case (Algorithm 9).

Algorithm 8 East Routing Logic of West In-Port

```
1: if OK(E) then
2:   W_to_E_req = 1;
3: else
4:   if OK(N1) ∧ (OK(N, E) ∨ OK(N, N1)) then
5:     W_to_N1_req = 1;
6:   end if
7:   if OK(S1) ∧ (OK(S, E) ∨ OK(S, N1)) then
8:     W_to_S1_req = 1;
9:   end if
10:  if W_to_N1_req = 0 ∧ W_to_S1_req = 0 then
11:    if cy = DIMY − 1 then
12:      Set intermediate destination to north
13:      W_to_SUB2_req = 1;
14:    else if cy = 0 then
15:      Set intermediate destination to south
16:      W_to_SUB2_req = 1;
17:    end if
18:  end if
19: end if
```

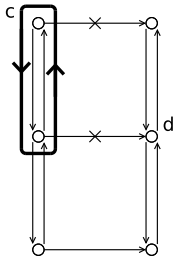


Figure 8.8: *Livelock scenario prevented by the routing logic. Processing nodes c and d are the current node and the destination node respectively. The crossed channels are faulty.*

When the packet has arrived in the correct column, it is routed south towards its destination without leaving the column. It can use both subnetworks. If it uses the current subnetwork only, deadlocks will occur. The packet leaves the column only if both the vertical channels leading towards the destination are faulty (Lines 7–14 of Algorithm 9).

8.3.2 Results

We have run the tool on meshes of different sizes (see Table 8.1). These results have been obtained on a Sun Fire X4440 machine, with four 2.3GHz Quad-Core AMD Opteron 8356 processors (16 cores in total) and 128 GB of memory. For each mesh, the routing function is proven deadlock-free, livelock-free, connected and correct in all faulty configurations.

Algorithm 9 South Routing Logic of West In-Port

```
1: if OK(S1) then
2:    W_to_S1_req = 1;
3: end if
4: if OK(S2) then
5:    W_to_SUB2_req = 1;
6: end if
7: if W_to_S1_req = 0 ∧ W_to_SUB2_req = 0 then
8:    if OK(E) then
9:       req_east = 1;
10:   else
11:      Set intermediate destination to west
12:      W_to_SUB2_req = 1;
13:   end if
14: end if
```

Mesh	Number of configs	Time (hh:mm:ss)
8x8	73,536	00:01:07
10x10	179,700	00:05:35
12x12	372,816	00:22:07
14x14	690,900	00:58:58
16x16	1,178,880	04:30:24
18x18	1,888,596	09:59:16
20x20	2,878,800	22:51:21

Table 8.1: *Results with 2 faults*

We have also run the algorithm on a 4x3 mesh with more than two faults. We let the algorithm assemble quantitative information (see Table 8.2). As the number of faults increases, the number of deadlocks and livelocks increase. Note that these results hold for a relatively small mesh. In larger meshes, we expect the percentages to be lower, as there will be more separate faults which do not interact with each other.

Faults	3	4	5
Configs	59640	1,028,790	13,991,544
Topology violation	0%	0%	0%
Disconnected routing	0.07%	0.26%	0.64%
Deadlock	0.24%	0.89%	2.04%
Livelock	0.09%	0.31%	0.69%

Table 8.2: *Results with more than 2 faults in a 4x3 mesh.*

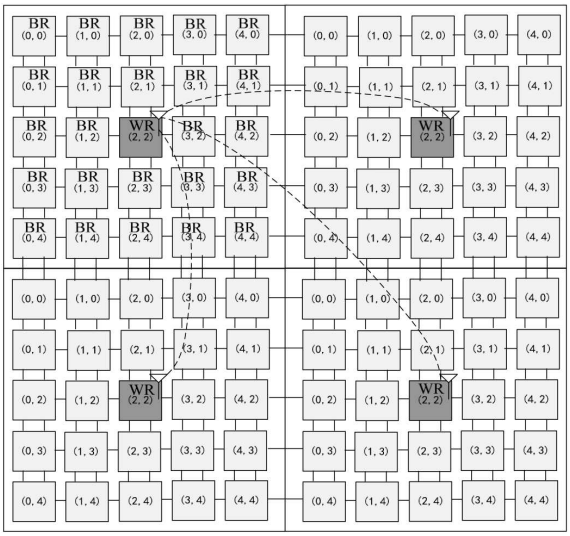


Figure 8.9: *Topology of Wireless NoC [144]*

8.4 NePA with Wireless Routers

Wang et al. design and analyze a hybrid NoC system where the standard wired on-chip communication coexists with on-chip *wireless* communication technology [144]. This hybrid structure is called Wireless NoC (WNoC).

Wireless communication can improve performance of the chip in terms of latency, throughput and power consumption. However, a sophisticated routing logic is required to determine which messages are sent wirelessly and which are not. We introduce the topology and the routing logic. For a more extensive explanation, and for a detailed analysis of performance and feasibility of wireless on-chip communication, we refer to the paper of Wang et al. [144].

Wang et al. constructed the WNoC by replacing some of the routers in the NePA architecture with wireless routers (WRs). The topology is split up into quadrants (see Figure 8.9). Each quadrant contains one wireless router connected to the processing node in the middle of the quadrant. Each wireless router can communicate with all three other wireless routers.

The WNoC contains deadlocks, if no additional measures are taken such as virtual channels or restrictions on the routing logic to prevent messages from using WRs under specific conditions. We have specified the WNoC without such additional constructs, and therefore found deadlocks.

8.4.1 Routing Logic

Algorithm 10 contains the routing logic of WNoC as formalized for our tool. Function NePA_routing formalizes the standard minimal adaptive routing in NePA. This function takes as parameters a current processing node and the destination and returns a set of next hops.

Algorithm 10 Routing Logic of WNoC

Require: Current node c;
Require: Destination d;
1: $H_W =$ distance to destination using WRs;
2: $H_B =$ distance to destination using wires only;
3: **if** $H_W < H_B - \delta \wedge \text{quadrant}(c) \neq \text{quadrant}(d)$ **then**
4: $d_{\text{WR}} =$ WR in current quadrant;
5: **if** $c \neq d_{\text{WR}}$ **then**
6: **return** NePA_routing(c, d_{WR})
7: **else**
8: Transmit wirelessly;
9: **end if**
10: **else**
11: **return** NePA_routing(c, d)
12: **end if**

Line 3 decides whether the message will use a WR. In the WNoC, the decision to transmit messages wirelessly is based on the distance from the current location to the destination. A WR is used if it takes significantly more steps to arrive at the destination using wires only than using WRs. Significance is expressed using some predefined but constant value δ: the higher δ, the more the WRs will be used. If the difference between the number of steps taken wirelessly and the number of steps taken using wires only is greater than δ, the routing logic will send the packet towards a WR.

If the message makes use of the WR, it is first routed towards it (Line 6). If it has arrived at the WR, it will be transmitted wirelessly to the quadrant of its destination (Line 8). From this point on, it will be routed towards its destination in the regular way (Line 11).

8.4.2 Results

Figure 8.10 gives an example of a deadlock configuration found by DCI2. We have set δ to 2. Each worm W is destined for processing node d_W. Worm A is injected at $(9, 3)$. The wired distance to its destination is 10 hops. The distance using WRs (in this case WR $(7, 2)$) is 8. The wireless distance is less than the wired distance, but for $\delta = 2$ the difference is not considered significant and the message is routed without making use of WRs. Worm A is blocked by worm B, which for similar reasons uses wires only. Worm B, on its turn, is blocked by worm C. Worm D blocks worm C. It is injected at $(4, 0)$ and destined for $(6, 9)$. The wired distance is 11, whereas the distance using WR $(2, 2)$ is 8. As $11 - 8 = 3$, the yield of using a WR is deemed significant. Worm D is first routed west to WR $(2, 2)$, routed wirelessly, and subsequently is routed towards its destination. Finally, worm A blocks worm D, completing the circular wait.

The circulair wait is not necessarily a deadlock, as routing is adaptive. In this configuration, all four worms have no escapes. The configuration is therefor a deadlock.

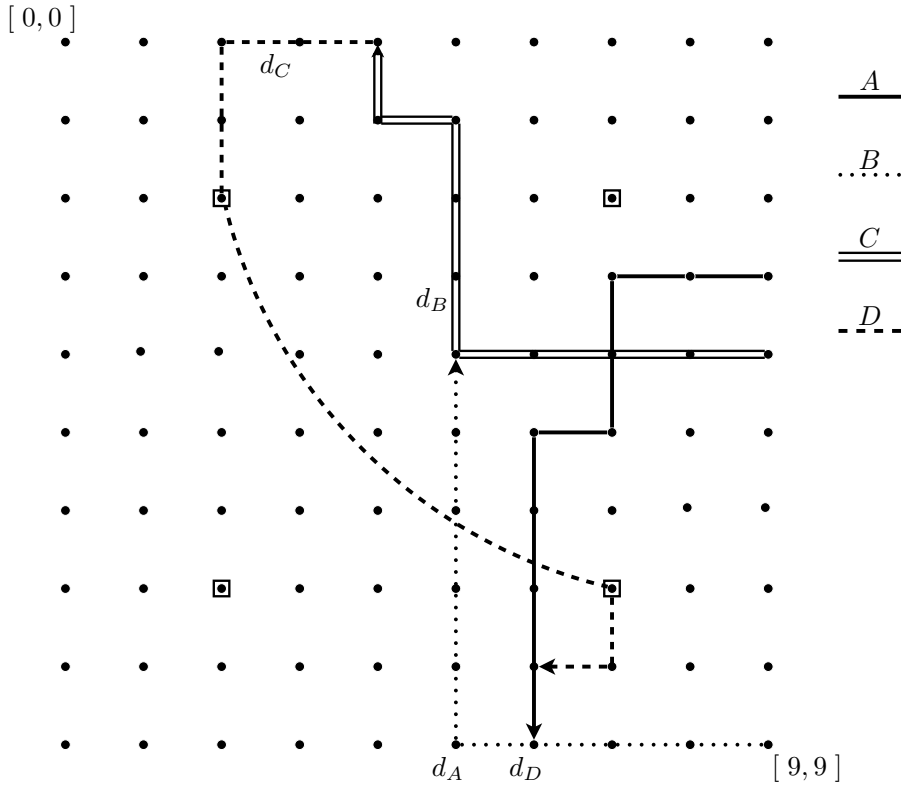


Figure 8.10: *Deadlock in WNoC. The squares are the WRs.*

We have defined the wireless routing logic for a 10x10 mesh. Deadlocks are found instantaneously. We have experimented with different values for δ. Any $\delta \leq 8$ yields a deadlock. For $\delta > 8$, the wireless routers are not used at all, meaning that the routing logic equals a deadlock-free minimal adaptive routing.

8.5 Comparison to Taktak et al.

Taktak et al. present the deadlock checking tool ODI [138] (see Section 7.8 at Page 129). The algorithmic complexity of their algorithm is $O(N^4)$, whereas our tool checks for deadlock-freedom of wormhole networks in $O(N^3)$. Figure 8.11 shows a comparison with our solution for the "Shortest Path with XY" routing function. This graph shows that the the theoretical improvement in algorithmic complexity is reflected in experimental results as well.

Zhang et al. present the application of ODI to a fault-tolerant routing function [149]. ODI is used to establish deadlock freedom of their design in all configurations with one fault in a 10x10 mesh. The total number of configurations that needed to be checked is about 800. We analyze our fault tolerant routing logic in all the configurations with two faults in a 20x20 mesh. This amounts to

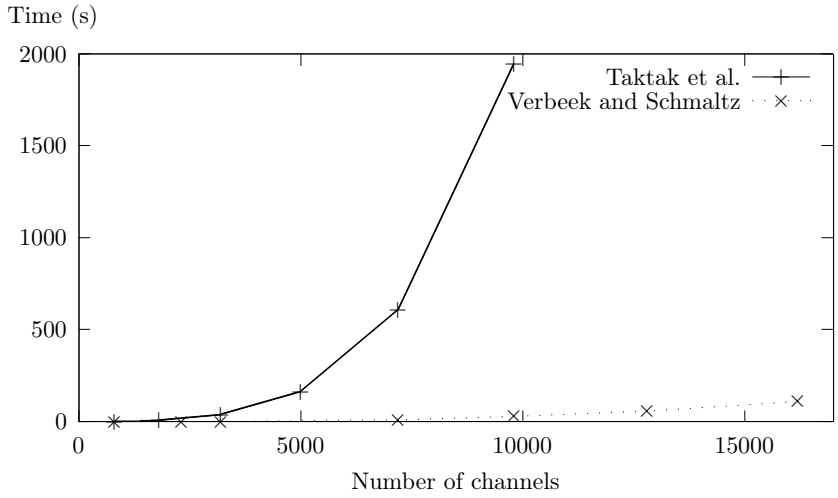


Figure 8.11: *Comparison to Taktak et al.*

checking 2,878,800 configurations. Additionally, DCI2 establishes livelock freedom and connectedness as well.

8.6 Conclusion

The tool DCI2 has been presented that checks for deadlocks, livelocks, topology violations and connectedness of the routing logic. Different faulty configurations are verified in parallel, so that deadlocks are found quickly. DCI2 can produce quantitative results on how many configurations are correct.

Basically, we have applied DCI2 as a debugging tool for programming routing logic. Initially, we ran the tool on the NePA topology with simple minimal adaptive routing logic. With two faults, the tool quickly found a configuration in which the routing function becomes disconnected. For example, when the source and destination are in the same row and an intermediary horizontal channel is faulty. After adding a case distinction to the routing logic to handle this situation, we reran the tool on the new routing logic. The change may cause a deadlock or livelock. It may route packets away from faults, but towards a situation where the destination is no longer reachable, effectively causing a disconnected routing function. Iteratively, we first add or remove case distinctions, then rerun the tool and analyze its output. We have repeated this process, until the routing logic was completely correct. As an example, the deadlock in Figure 8.7a was found by the tool. At this point in the design process, the second case distinction in Line 4 of Algorithm 7 had not been added.

The irregular routing logic of the WNoC is hard to verify manually. DCI2 finds deadlocks instantaneously in a 10x10 mesh. Because a result is quickly produced, we could experiment with different values of δ, i.e., with more or less usage of the wireless routers. Deadlocks have been found for all non-trivial values of δ.

The properties verified by DCI2 correspond to proof obligations required to

prove productivity of communication networks (see Chapter 3). Checking for topology violations effectively checks that the routing function is correctly typed, i.e., it has type $P \times P \mapsto \mathcal{P}(C)$. Proof Obligation 1 states the routing logic must be connected, which is checked by DCI2. Deadlock freedom (Proof Obligation 7) is discharged using the formally verified algorithms presented in the previous chapter. Checking for absence of livelocks discharges Proof Obligations 11 and 12. DCI2 discharges all proof obligations that concern the routing logic.

In Chapter 4 we presented a full proof of correctness of a communication network with west-first routing in a two-dimensional mesh. This proof is done once and for all, as the size of the mesh is left parametric. The proof effort took approximately one week of ACL2 interaction. In contrast, DCI2 requires 1.49 seconds to establish deadlock freedom of the same network for a 65 by 65 mesh. DCI2 is a push-button alternative for discharging Proof Obligation 7, but works for bounded instances only.

DCI2 provides the possibility to use formal methods during the design process of a communication network. Assuming starvation freedom and correctness of injection, any packet or wormhole network that passes a check by DCI2 is productive.

Part IV

Integrated Network Layer Deadlock Verification

Chapter 9
Microarchitectural Deadlock Verification

The previous part dealt with verification of the network layer in isolation. In this part, we focus on the monolithical verification of deadlock freedom. The integrated network model takes details into account of both the applications running on top of the network and the transfer protocol. We address two issues. First, we require a formal language that is sufficiently expressive to model the behavior of all three layers. The difficulty lies in the fact that the possible behavior is unrestricted, e.g., cache coherency protocols, master/slave protocols, credit-based flow controls, broadcasts, etc. A language is required that is on one hand sufficiently expressive, but on the other hand restricted and formal enough to allow efficient deadlock detection. Our solution is to not define one language, but a family of languages. This allows defining custom microarchitectural components, resulting in a custom Microarchitectural Description Language (MaDL). The second problem is how to detect deadlocks efficiently. We define a deadlock detection algorithm parameterized with the language in which the communication network is defined. An efficient, optimized and tailored deadlock detection algorithm is obtained for each language in the family. Using SMT solvers, linear programming solvers, and invariant generation the algorithm can handle message dependencies, counters, virtual channels, parametric buffer sizes, and many other aspects of microarchitectural models.

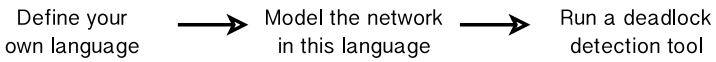


Figure 9.1: *Work flow enabled in this chapter.*

The results in this chapter are part of work in progress. We have not yet formally proven correctness of the theorems in the ACL2 theorem prover. The examples are mostly artificial and academic. We will clearly point to restrictions and limitations of our approach.

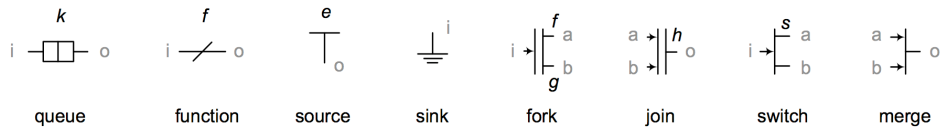


Figure 9.2: *Eight primitives of the xMAS language. Italicized letters indicate parameters. Gray letters indicate ports.*

9.1 MaDLS

This section describes a family of languages. For each language in the family, we can detect deadlocks efficiently. This family is based on the language xMAS (eXecutable MicroArchitectural Specification), developed by Intel [24]. The xMAS language is a graphical language consisting of eight primitives. It can be used to model a communication network at the microarchitectural level. We first present xMAS. Then, we provide a formal way of defining custom primitives. This yields the family of languages. We provide several examples to show the expressivity of this family.

9.1.1 xMAS: a MaDL for communication fabrics

An xMAS model is a network of primitives connected via channels. A channel is connected to an *initiator* and a *target*. Each channel consists of three *signals*. Channel signal $x.irdy$ indicates whether the initiator is ready to write to channel x. Channel signal $x.trdy$ indicates whether the target is ready to read channel x. Channel signal $x.data$ contains data that is transferred from the initiator output to the target input if and only if both signals $x.irdy$ and $x.trdy$ are set to true.

Figure 9.2 shows the eight primitives of the xMAS language. A *queue* stores data and is the state holding element. 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 message types and represent message dependencies inside the fabric or in the model of the environment. Messages are non-deterministically produced and consumed at *sources* and *sinks*. A source may process multiple message types. 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 combined. 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 execution semantics of an xMAS network consists of a combinatorial and a sequential part (Figure 9.3). The combinatorial part updates the values of channel signals. The sequential part is the synchronous update of all queues according to the values of the channel signals. A simulation cycle consists of a combinatorial and a sequential update. A sequential update only concerns queues, sinks, and sources. We denote these primitives as *sequential primitives*. Other primitives are

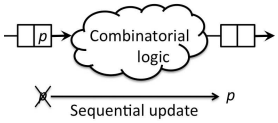


Figure 9.3: *Sequential updates regulated by combinatorial primitives*

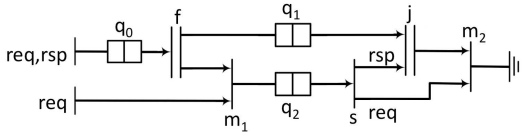


Figure 9.4: *Microarchitectural model*

denoted as *combinatorial*.

A *configuration* represents the current occupation of queues, i.e., the current state. Configurations are updated when messages are produced, consumed, or moved to one or more next queues. Similar to the previous part, we assume that typing information is known. For each queue q, we assume that $\tau(q)$ returns the set of packets that can be in queue q.

Example 9.1 Consider the xMAS model in Figure 9.4. Assume two request packets are injected in queue q_0 and the other source remains silent. At the fork, the combinatorial semantics propagates to queues q_1 and q_2 a positive *irdy* signal. As queues q_1 and q_2 are ready to receive, a positive *trdy* is propagated back to q_0. In the next sequential update, one request packet is moved to queues q_1 and q_2. The remaining packet will be moved in the next simulation cycle. Requests in q_2 are routed to the sink and are eventually consumed. The requests in q_1 are blocked as no response arrives at the join. Indeed, responses injected in q_0 are blocked by the requests in q_1 if q_1 is full.

Figure 9.4 is a *formal* description of a communication fabric. For each primitive, the exact semantics are defined by equations on the signals. These equations assign a value to the *irdy* signals of the channels going out of the primitive. This value represents whether the primitive is ready to transmit a value over that outgoing channel. The equations also assign a value to the *trdy* signal of each incoming channel. This value represents whether the primitive is ready to receive the data at the in-channel. Lastly, the primitive semantics assign a value to the *data* signals of all outgoing channels.

Running Example 9.1 The semantics of the *switch* are defined as follows:

$$
\begin{aligned}
o_1.irdy &::= i.irdy \text{ and } s(i.data) \\
o_2.irdy &::= i.irdy \text{ and not } s(i.data) \\
i.trdy &::= (o_1.irdy \text{ and } o_1.trdy) \text{ or } (o_2.irdy \text{ and } o_2.trdy) \\
o_1.data &::= i.data \\
o_2.data &::= i.data
\end{aligned}
$$

Output channel o_m ($m \in \{1, 2\}$) is ready to transmit if and only if the initiator of

the input channel is ready to transmit towards o_m and if the packet at the input channel is routed towards channel o_m. Input channel i is ready to receive if and only if one of the output channels is ready to transmit. Data at the outputs is copied from the input.

9.1.2 A Family of MaDLs

Intel published the eight primitives in Figure 9.2. Many other useful primitives can be imagined, such as virtual channels, scoreboards, out-of-order queues, and adaptive switches. Some of these can be constructed in terms of the eight basic primitives, but some, e.g., the adaptive switch, cannot. We define a core language called xMAS_0 that contains only sources, queues and sinks. To obtain a custom language, xMAS_0 is extended with user-defined custom primitives.

To create a custom primitive, the semantics of this primitive are needed. These semantics can be described in the syntax presented in Figure 9.5. The syntax allows the use of functions and the Boolean connectives *and*, *or* and *not*. Function names are left uninterpreted. Quantifiers may be used to express semantics of primitives with a parametric number of inputs and outputs. Input channel signals $i.irdy$ and $i.data$ and output channel signal $o.trdy$ may be used. Signal $i.select$ may be used to determine which in-channel gets its turn in case of multiple in-channels. We use this signal to abstract away from the arbitration policy applied by the primitives. The semantics is such that at most one channel is selected, i.e., if $i.select$ is true for some input channel i, it implies that $i'.select$ is false for all other input channels i'. The channel name i_{sel} is reserved to refer to the currently selected channel.

prim_semantics	:=	| prim_semantics signal ::= sign_semantics
sign_semantics	:=	(sign_semantics *and* sign_semantics)
		| (sign_semantics *or* sign_semantics) |
		| *forall name* sign_semantics | *exists name* sign_semantics
		| *not* sign_semantics | *name*(signal) | signal
signal	:=	*name.irdy* | *name.trdy* | *name.data* | *name.select*

Figure 9.5: *Syntax in which primitive semantics can be defined*

Primitive semantics will be written in *italic* font. We let $s \in S$ denote that string s is a line in some semantics S. For example, if S refers to the semantics of the switch then $o_2.data ::= i.data \in S$.

The core language xMAS_0 is defined as the language that contains the source, sink and queue:

$$\mathsf{xMAS}_0 \stackrel{\text{def}}{=} \{queue, source, sink\}$$

The original xMAS language can be obtained using our core language and syntax, i.e., one can define $\mathsf{xMAS} = \mathsf{xMAS}_0 \cup \{switch, merge, join, fork, f\}$ using the semantics given by Chatterjee et al [24].

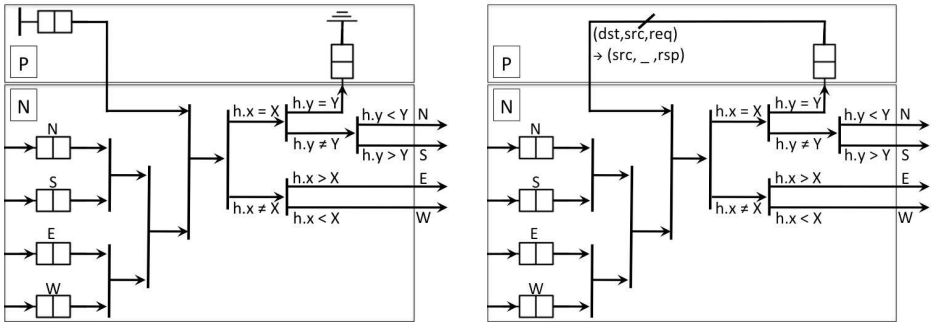
9.1.3 Examples

2D Mesh with message dependencies

Section 1.3.1 (see Page 10) presents an example of a communication network with message dependencies. A set of masters send out requests to slaves. Upon receiving a request, a slave sends a response back to the master. The masters and slaves are laid out in a 2D mesh with XY routing. Some layout dictates which nodes are masters and which are slaves.

Figure 9.6 shows the xMAS model of this network. Channels are modelled with queues. The processing nodes containing the routing logic are modelled in area N. Packets arriving at a processing node are routed either into the network according to the XY routing logic or sent to a local buffer storing the arrived packets.

The master/slave protocol is modelled in area P. Each master injects messages into the local-in queue (see Figure 9.6a). Packets of a master with coordinates (x, y) are restricted to the form (s, d, t) where $s = (x, y)$, $d = (x', y')$ such that (x', y') denotes a slave, $(x, y) \neq (x', y')$ and $t = req$. The source is thus typed accordingly. If a response arrives at the local-out queue, we assume it can eventually be consumed. It is therefore switched to a sink. The slaves do not inject or consume messages (see Figure 9.6b). If a request arrives, it is not consumed but altered into a response with as destination the source of the request. It is then switched into the interconnect according to the routing logic.



(a) *Master* **(b)** *Slave*

Figure 9.6: *Processing nodes with XY routing (area N) and master/slave protocol (area P).*

Spidergon with write-invalidate cache coherency

We define the routing logic of a Spidergon processing node with shortest path routing (see Figure 9.7). Additionally, a cache coherency protocol is inserted between process P and processing node N. The protocol is a write-invalidate protocol with snooping [136]. Processes are either processors or memory modules. Packets are of type read or write indicating that the process performs a request to read from or write to a memory module. Broadcasting occurs on a separate network, which connects all caches in a broadcast ring.

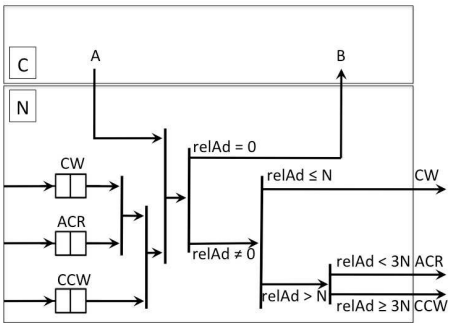


Figure 9.7: *Processing node of Spidergon with shortest path routing*

The protocol is defined as follows. Each cache can either be `valid` or `invalid`. If a read arrives at a `valid` cache, the corresponding cache line is checked. If the cache did not contain the requested data, the read is forwarded to the interconnection network to be directed to the memory module. If a read arrives at an `invalid` cache it is blocked until the cache is `valid`. If a write arrives at a `valid` cache, it changes the state of the cache to `invalid` and broadcasts the write to all other caches, invalidating them as well. When all caches are `invalid`, the write is performed synchronously after which all caches are set to state `valid`.

Figure 9.8 shows the xMAS specification of the cache model C for some process P connected to processing node N. A read-packet is routed towards a queue buffering the reads. If the lock is available, a read can proceed. Otherwise it is blocked until the lock is available. A function primitive models the reading of the corresponding cache line. In case of a cache hit, the packet is returned to the process immediately. Otherwise it is routed through the network to the corresponding memory module.

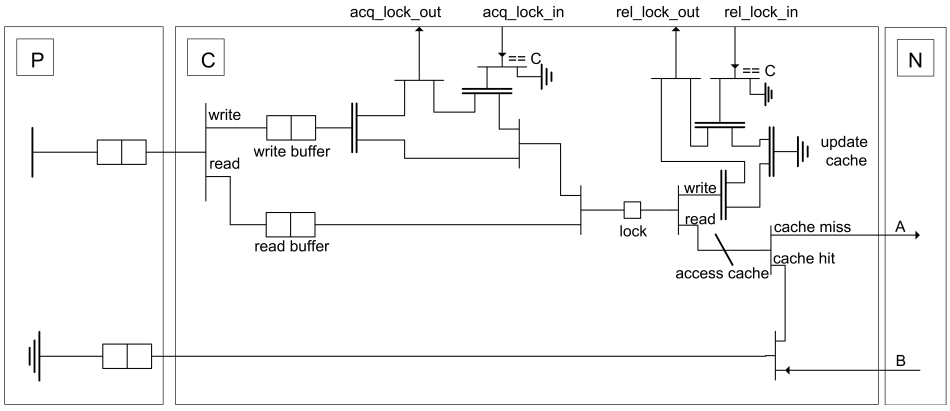


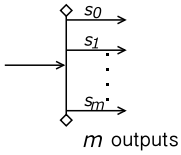
Figure 9.8: *xMAS model of cache coherency protocol*

A write packet is routed towards a queue buffering the writes. The fork duplicates the write: one is sent to the lock of the current cache. The second packet

is sent to the next cache in the broadcast ring through the `acq_lock_out` line. It enters the next cache through the `acq_lock_in` line. Here it is again duplicated, so that it can again lock the current cache and be forwarded to the next cache. This duplication proceeds until a write arrives back at the `acq_lock_in` line. In this case, the entire ring has received the write and thus all the forks have duplicated exactly one write for each lock. All caches are locked. The write is routed from the lock to a second fork. One write is sent to the join, one is sent to the join of the next cache. The joins ensure that all locks on the broadcast ring are released synchronously. The actual writing that occurs when a write packet is sent is represented by a sink.

West-first routing in a 2D mesh

So far, we have used traditional primitives only. In this example we define a custom primitive, namely the adaptive switch. This switch has one input and m outputs (see Figure 9.9a). Figure 9.9b shows its semantics. The adaptive switch can receive a packet from its input channel i if there exists an outgoing channel o_m at which a transfer can occur. Each outgoing channel o_m is ready to transmit if the packet at channel i can be routed towards channel o_m. Data is simply transferred from the input towards the output.



(a) *Primitive*

$$
\begin{array}{lcl}
i.trdy & ::= & exists\ m \cdot (o_m.irdy\ and\ o_m.trdy); \\
o_m.irdy & ::= & (i.irdy\ and\ s_m(o_m.data)); \\
o_m.data & ::= & i.data;
\end{array}
$$

(b) *Semantics*

Figure 9.9: *Adaptive Switch*

Using the adaptive switch, we have defined a 2D mesh topology with west-first routing [61]. Figure 9.10 shows a processing node with coordinates (X, Y). An incoming packet p is analyzed. We assume its destination coordinates are $(p.X, p.Y)$. Packets are only routed westwards, if the packet has just been injected or if the packet was already heading for the western direction. Packets can always go east if the destination is east of the current processing node. Similarly, packets can always go north if the destination is north. However, turns from south to north are prohibited. Section 4.2 contains more details on west-first routing.

We have introduced a family of languages. Each language consists of the three core primitives source, sink and queue with some additional custom primitives. For each custom primitive, semantics have to be specified that determine when and how the primitive transfers a packet from its inputs to its outputs.

9.2 Deadlock Detection Algorithm

In this section, we define a deadlock detection algorithm that takes as parameter not only the communication network in which deadlocks are to be found, but also

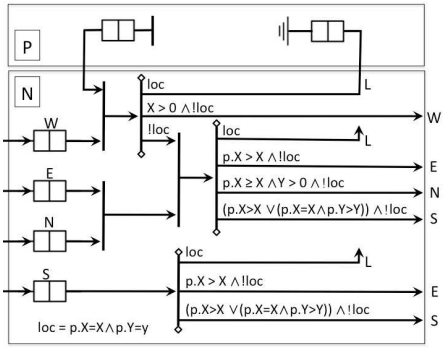


Figure 9.10: *Processing node with west first routing*

the language in which the network is defined. A deadlock is a situation in which some channel permanently wants to send a packet, but cannot transmit it. Such a channel is called a *dead* channel. The algorithm presented in this chapter searches for such dead channels. More specifically, it searches for a reachable configuration in which a channel is permanently blocked but not permanently idle. The major purpose of the algorithm is to reduce the search for a reachable configuration to the search for a solution of a linear integer arithmetic problem (LIAP). This problem has been well-studied and many different efficient tools exist to deal with them [49, 40, 6]. The reduction makes use of *blocking* and *idle formulas* for each core primitive and each custom primitive. These formulas express necessary and sufficient conditions under which channels are permanently blocked or idle. We automatically generate these formulas from the semantics of the custom primitives and use them in our deadlock detection algorithm.

First, we recapitulate the formal definition of deadlock provided by Gotmanov et al. [65]. Then we explain by example how the search for a reachable deadlock is reduced to the search for a solution of a LIAP. Finally, we show how blocking and idle formulas of custom primitives are used in our deadlock detection algorithm.

9.2.1 Definition of Deadlock

A dead channel c has – ins ome reachable configuration – its $c.irdy$ signal set to true and its $c.trdy$ signal *stuck-at* false. Formally, the LTL definition of a dead channel is as follows:

$$\mathbf{Dead}(c) = \Diamond(c.irdy \cdot \Box\, \neg c.trdy)$$

Example 9.2 In Example 9.1 above, output channel o of queue q_1 is dead. Consider the execution in which k requests are injected by the first source, with k the size of queue q_1 (see Figure 9.11). In this execution, signal $o.irdy$ is set to true as there is a packet at the head of the queue. Signal $o.trdy$ is stuck-at false as the join will never receive a response on its second input.

Gotmanov et al. simplify this definition of deadlock using *persistency* [65]. A channel is persistent if its signals – once they are set – are stable until a transfer

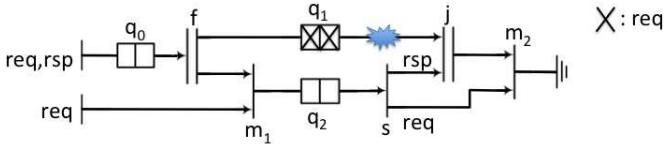


Figure 9.11: *Microarchitectural model*

occurs. Gotmanov et al. prove that xMAS networks are persistent. This allows them to express deadlocks in terms of *blocked* and *idle* channels.

Definition 9.1
A channel is *permanently blocked*, notation $\mathbf{Block}(c)$, if and only if eventually its *trdy* signal is permanently low.

$$\mathbf{Block}(c) \stackrel{\text{def}}{=} \Diamond\Box(\neg c.trdy)$$

A channel is *permanently idle*, notation $\mathbf{Idle}(c)$, if and only if eventually its *irdy* signal is permanently low.

$$\mathbf{Idle}(c) \stackrel{\text{def}}{=} \Diamond\Box(\neg c.irdy)$$

A channel c is *dead*, notation $\mathbf{Dead}(c)$, if and only if it is blocked and not idle.

$$\mathbf{Dead}(c) \stackrel{\text{def}}{=} \mathbf{Block}(c) \wedge \neg\,\mathbf{Idle}(c)$$

9.2.2 Deadlock LIAPs

We formulate LIAPs in which there exists a variable for each queue q and each packet p that can be in queue q. The LIAPs are constructed in such a way that a solution assigns to each queue a number of packets in such a way that the resulting configuration is a deadlock. If no solution exists, there is no deadlock. The value of variable $\#q.p$ represents the number of packets of type p in queue q. The value of variable $\#q$ stores the total number of packets in the queue. We assume that for each queue q, the size of q can be accessed with the variable $q.size$.

Example 9.3 Consider again the deadlock in Figure 9.11. In this deadlock configuration, requests are assigned to queue q_1. The number of requests in queue q_1 is represented by variable $\#q_1.req$ which is equal to $q_1.size$ in any deadlock. Queue q_2 does not contain responses. Therefore, $\#q_2.rsp = 0$.

For each queue q and for each packet $p \in \tau(q)$, a LIAP is created. We provide the LIAP for queue q_1 and packet req for the network in Figure 9.11.

If a request is permanently blocked in queue q_1, then there must be at least one request in this queue. The following constraint is added to the LIAP:

$$\#q_1.req \geq 1 \qquad \text{(Request in } q_1\text{)}$$

The join blocks this packet only if no response arrives at its second input. In order for this to happen, queue q_2 may not contain responses.

$$\#q_2.rsp = 0 \qquad \text{(No responses in } q_2\text{)}$$

In order for q_2 to *permanently* be idle for responses, queue q_1 must be full. A third constraint is induced:

$$\#q_1 = q_1.size \qquad (q_1 \text{ is full})$$

The LIAP assembled so far has many solutions. The deadlock described above is one of them, but additional constraints are required to rule out bogus solutions. For example, a bogus solution might assign three packets to queue q_2 even when the size of the queue is only two. For each queue q, constraints are added to the LIAP to rule out illegal configurations.

$$\#q \leq q.size \qquad \text{(Legality constraints)}$$

Queue q_0 can contain requests and responses. To enforce that it cannot contain other packets and to enforce a link between the variables $\#q.p$ and $\#q$, constraints of the following form are added:

$$\#q = \sum_{p \in \tau(q)} \#q.p \qquad \text{(Typing constraints)}$$

A solution to the LIAP assembled so far yields a *legal* deadlock configuration. However, this configuration is not necessarily *reachable* from the initial configuration. For example, a solution might fill queue q_1 with responses and leave queue q_2 empty, even though the corresponding configuration is not reachable. Invariants are added to the LIAP to rule out unreachable configurations. In our example, the number of responses in q_1 is always equal to the number of responses in q_2.

$$\#q_1.rsp = \#q_2.rsp \qquad \text{(Invariant)}$$

The LIAP obtained this way has multiple solutions. All of these solutions are legal and reachable deadlocks. The configuration described in Example 9.3 is one of them.

9.2.3 Algorithm Paraphernalia

The algorithm consists of two parts: function BlockDetect determines whether a channel c can be permanently blocked, and function IdleDetect determines whether some channel can be permanently idle. To detect a dead channel, we execute:

$$\text{DeadDetect} \equiv \neg\text{IdleDetect} \wedge \text{BlockDetect}$$

Both functions take as parameters a channel c that is to be explored, the current packet p, the communication network N, and the language L in which network N is defined. They return a LIAP that is given to an SMT solver.

As shown in the previous example, formulating the LIAP requires the following:

- For a queue q and a packet p, constraints must be computed that reflect the permanent blocking of p in q;
- Legality and typing constraints must be computed to rule out illegal configurations;

- Invariants must be computed to rule out unreachable configurations.

The legality and typing constraints can easily be computed. A technique generating the invariants has been presented by Chatterjee et al. [22, 23]. The issue to be tackled is to determine efficiently which constraints have to be added to represent that a packet is permanently blocked. Different combinatorial primitives induce different conditions under which channels are permanently blocked or idle. Since we allow custom combinatorial primitives, we have to be able to derive the desired behavior of the algorithm from the user-defined semantics of these primitives.

For the three basic primitives in xMAS_0, the behavior of the algorithm is fixed. For custom primitives, the behavior of the algorithm is *not* fixed. We use blocking and idle formulas as an intermediary between primitive semantics and the behavior of the algorithm [65]. Blocking and idle formulas represent the exact conditions that have to be satisfied for a channel to be permanently blocked or idle.

Running Example 9.2 For packet p, the blocking formula for the switch is defined as follows:

$$\mathbf{Block}(i) \equiv \texttt{if } s(p) \texttt{ then } \mathbf{Block}(o_1) \texttt{ else } \mathbf{Block}(o_2)$$

A packet p is permanently blocked by a switch if and only if the channel to which the packet is routed is permanently blocked. The idle formulas for the switch are defined as follows:

$$\begin{aligned} \mathbf{Idle}(o_1, p) &\equiv \mathbf{Idle}(i, p) \texttt{ or not } s(p) \\ \mathbf{Idle}(o_2, p) &\equiv \mathbf{Idle}(i, p) \texttt{ or } s(p) \end{aligned}$$

An outgoing channel of a switch is permanently idle for packet p if and only if either packet p never arrives at the switch or packet p is not routed towards the channel.

The idle formula of the switch directs the algorithm to recursively compute whether the input channel can be permanently idle, to compute whether the packet is routed towards the current channel, and to compute the disjunction of both results. As another example the blocking formula of a join tells the algorithm that in order to establish whether a join can be permanently blocked it must perform both a forward search to establish that the output of the join can be permanently blocked and a backward search to determine that the other input can be permanently idle. The essence of blocking and idle formulas is that they completely determine the desired behavior of our algorithm.

For each custom primitive, the semantics determine the conditions under which the primitive is permanently blocked or idle. Thus the semantics determine the blocking and idle formulas and consequently, the semantics determine the behavior of the algorithm for custom primitives. Figure 9.12 presents an overview.

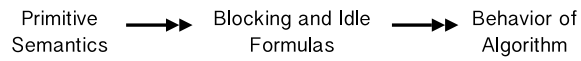


Figure 9.12: *From semantics of primitives to behavior of the algorithm*

We have created an automatic translation from primitive semantics to blocking and idle formulas. We postpone details on this translation to Section 9.3. The translation from primitive semantics to blocking formulas is denoted $[\![]\!]^p_{\mathbf{B}}$. The translation to idle formulas is denoted $[\![]\!]^p_{\mathbf{I}}$. Our translations satisfy the following theorems:

$$\begin{array}{lcl} \texttt{not}[\![c.trdy]\!]^p_{\mathbf{B}} & \iff & \mathbf{Block}(c) \\ \texttt{not}[\![c.irdy]\!]^p_{\mathbf{I}} & \iff & \mathbf{Idle}(c,p) \end{array}$$

If one looks up the semantics of the *trdy* (*irdy*) signal of some channel and translates the semantics to a blocking (idle) formula, this formula returns `false` if and only if the channel is permanently blocked (idle).

Running Example 9.3 The primitive semantics of the switch are translated to the following blocking formula:

$$\texttt{not}([\![i.trdy]\!]^p_{\mathbf{B}}) \quad \equiv \quad (\texttt{not } s(p) \texttt{ or } \mathbf{Block}(o_1)) \texttt{ and } (s(p) \texttt{ or } \mathbf{Block}(o_2))$$

The code for the three basic primitives is fixed. The code for custom primitives is automatically generated from blocking and idle formulas. Given a formula F, the generated code is denoted by "F". Boolean connectives are translated to special C functions DISJUNCT, CONJUNCT and NEGATE. As we quantify over finite domains, quantifiers are translated to for-loops of con(dis)junctions. Occurrences of **Block** and **Idle** are translated to recursive calls to BLOCKDETECT and IDLEDETECT.

Running Example 9.4 Based on the blocking formula yielded by the translation in Running Example 9.3, the following code is executed by the algorithm to determine whether a switch x can be permanently blocking.
That is, "$\texttt{not}[\![c.trdy]\!]_{\mathbf{B}}$" denotes the following code:

```
1: ret_0 = DISJUNCT(NEGATE(s(p)), BLOCKDETECT(x.o_1, p, N, L)
2: if ret_0 then
3:    ret_1 = DISJUNCT(s(p), BLOCKDETECT(x.o_2, p, N, L)
4: else
5:    ret_1 = FALSE
6: end if
7: return CONJUNCT(ret_0, ret_1)
```

The generated code is compiled so that it can be executed by the algorithm. For sake of presentation, we assume an *eval* function that takes a piece of C code, compiles it, and executes it.

9.2.4 Deadlock Detection Algorithm

Algorithm 11 shows the pseudo code of our algorithm for determing whether a channel can be permanently blocked. Let the target of the current channel be a queue q. In order to be blocking, queue q must be full. Formula $\#q = q.size$ is the default return value (Line 2). If the current queue has not been visited yet, it is marked as visited (Line 4). For each packet $p' \in \tau(q)$, the algorithm recursively determines the formulas representing whether the next primitive can be permanently blocking (Lines 5–7). Queue q is permanently blocking if for one of the packets p' the corresponding formula is feasible. The algorithm disjunctively adds to the return value the conjunct of formula $\#q.p' \geq 1$ and the return value of

Algorithm 11 BLOCKDETECT(c, p, N, L)

```
1:  if c.trgt is a queue named q then
2:    ret := #q = q.size
3:    if ¬visited_B[q] then
4:      visited_B[q] = true
5:      for all p' ∈ τ(q) do
6:        ret \/= #q.p' ≥ 1 ∧ BLOCKDETECT(q.out, p', N, L)
7:      end for
8:      visited_B[q] = false
9:    end if
10:   return ret
11: else if c.trgt is a sink then
12:   return false
13: else if c.trgt is a source then
14:   \* Will never occur *\
15: else if c.trgt is a custom primitive in L named x then
16:   return eval("not⟦c.trdy⟧^p_B")
17: end if
```

the recursive call. Subsequently, it marks the current queue as unvisited (Line 8). A sink is never permanently blocked, as we assume fair sinks (Line 12). If the target of the channel is some custom primitive x, then the behavior of the algorithm is determined by the corresponding blocking formula (Line 16). The semantics of primitive x are translated into a blocking formula. C code that unfolds this formula is automatically generated and executed.

Algorithm 12 shows the pseudo code of our algorithm for determining whether a channel can be permanently idle. Let the initiator of the current channel be a queue q. A queue can be permanently idle for packet p for two reasons: either it does not contain packet p and its input is permanently idle for packet p (Line 4), or another packet p' is permanently blocked at the head of the queue (Line 7). A source is permanently idle for packet p if and only if packet p is not injected at the source, as we assume fair sources (Line 15).

Algorithms 11 and 12 provide the main structure of the algorithms. Many optimizations and features are added:

- Before running the algorithm, the typing information needs to be computed, i.e., we need to compute $\tau(q)$ for all queues q. To obtain this information we perform exhaustive simulations. For each source and for each possible packet p injected at the source, we simulate the injection in an empty network until it is consumed. During this simulation packet p is added to $\tau(q)$ for each visited queue.

  Consider the network in Figure 9.13. The network is deadlock-free. To establish this, it must be established that always eventually a packet "5" arrives at queue q_0. During the simulation of packet "0" in source src_0, queue q_0 is visited 6 times. This establishes that $\tau(q_0) = \{0, 1, \ldots, 5\}$. Using this information, our algorithm needs to visit queue q_0 just once to establish

Algorithm 12 IdleDetect(c, p, N, L)

```
1: if c.init is a queue named q then
2:   if ¬visited_I[q][p] then
3:     visited_I[q][p] = true
4:     ret := #q.p = 0 ∧ IdleDetect(q.in, p, N, L))
5:     visited_B[q] = true
6:     for all p' ∈ τ(q) such that p' ≠ p do
7:       ret \/= #q.p' ≥ 1 ∧ BlockDetect(q.out, p', N, L)
8:     end for
9:     visited_B[q] = false
10:    visited_I[q][p] = false
11:  end if
12:  return ret
13: else if c.init is a sink then
14:  \* Will never occur *\
15: else if c.init is a source named s then
16:  return p ∈ τ(s)
17: else if c.init is a custom primitive in L named x then
18:  return eval("not⟦c.irdy⟧_I^p")
19: end if
```

deadlock freedom of the network.

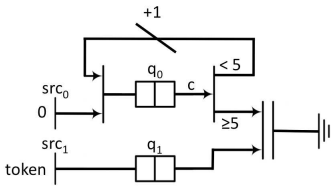


Figure 9.13: *xMAS model*

- The forwards exploration in BlockDetect (Line 5) creates a disjunction for each packet $p' \in \tau(q)$. The number of different types of packets can be large. For example, in a 16x16 2D mesh with masters and slaves, packets consist of 17 bits. This would require 2^{17} recursive calls. Many of the packets cause the exact same routing behavior. The algorithm automatically classifies packets that cause equal routing behavior into equivalence classes and performs at most one recursive call per equivalence class. For example, in the 2D mesh, a packet in a channel can be routed to at most five different other channels (North, South, West, East and Local). Instead of performing 2^{17} recursive calls, only 5 are necessary.

- At all times, the algorithm keeps track of the path leading from the initial queue to the current primitive. This path is always a conjunction of linear constraints, i.e., a linear program. Each modification of the path is checked

for consistency using the linear programming solver lp_solve. This way, unnecessary unfolding of primitives is kept to a minimum.

- At each combinatorial primitive, either disjunctions, conjunctions or combinations of both are unfolded. At all times, the algorithm keeps track of the total number of unfolded *con*junctions. If the algorithm encounters a visited queue, it checks whether this number is equal to zero. If this is the case, all conjunctions necessary to create a deadlock are unfolded. The current formula is sent to an SMT solver (we have used Yices [49]). If a solution is found, a deadlock is returned.
- We support standard optimizations for Boolean connectives, e.g, for a conjunction the second argument is not evaluated if the first argument evaluates to false. Expensive recursive calls are always the last arguments of disjunctions and conjunctions.
- The algorithm keeps track of which formulas are sent to Yices. If at any time a formula is subsumed by an earlier call to Yices, then this result is reused instead of performing a new Yices call.
- The algorithm can deal with the symbolic packet `ANY`, indicating that the primitive must be permanently idle for any packet.

9.2.5 Restrictions

Many custom primitives can be described using the syntax in Figure 9.5. However, some limitations on the syntax are necessary.

1. The network may not contain combinatorial cycles. This ensures termination.
2. If a primitive has multiple in-channels, we assume that it fairly selects which of these gets its turn, i.e., we assume that all primitives ensure starvation freedom. Without this restriction packets that are starving can permanently block other packets, resulting in new deadlocks.
3. A primitive selects an in-channel only if its initiator is ready.
4. We assume the semantics have been De Morganized as much as possible.
5. The negation cannot be applied to *irdy*, *trdy* and *select* signals. Our blocking and idle equations can only be expressed in terms of **Block** and **Idle**. To translate, e.g., $\neg i.irdy$, we need to establish $\Box \Diamond \neg i.irdy$, for which we must establish $\Diamond \Box i.irdy$. Thus, we must establish that channel i is at some point always able to transmit, i.e., it is *eager*. Checking efficiently for eagerness is still an open question.
6. Functions can take one *data* signal as input only. To obtain typing information, we simulate the injection of packets in isolation, i.e, we simulate the injection of each packet at each source in an empty network. If functions can take multiple packets, this way of obtaining typing information is no longer accurate.

A consequence of these restrictions is that we cannot express, e.g., a priority merge in our syntax. Such a merge gives priority to input i_H over the other input i_L. The semantics of $i_L.trdy$ require the negation of $i_H.irdy$. Another desired primitive that we cannot handle is the unrestricted join, which takes its two inputs and computes an output based on both inputs.

The current algorithm heavily relies on invariants to rule out unreachable deadlocks. The stronger the invariants, the less false deadlocks. We present some examples of false deadlocks found by our tool.

Asynchronous deadlocks Neither our tool nor the invariant generation takes into account that packets are moved synchronously throughout the network. The detected deadlocks are deadlocks under asynchronous semantics, but not necessarily under synchronous semantics. Consider the network in Figure 9.14. One packet is duplicated into two cycles. Both tokens move around in their cycles until at some clock cycle simultaneously the upper token reaches queue q_1 and the lower token reaches q_2. If this happens, both tokens are removed from the cycles and send into a deadlock. Under synchronous semantics, this never occurs as the tokens move in lock step fashion: they are either simultaneously in queues q_0 and q_2 or in queues q_1 and q_3. Under asynchronous semantics, the deadlock may occur. Our algorithm returns this deadlock. To exclude this deadlock, invariants are required dealing with the synchronizity between queues.

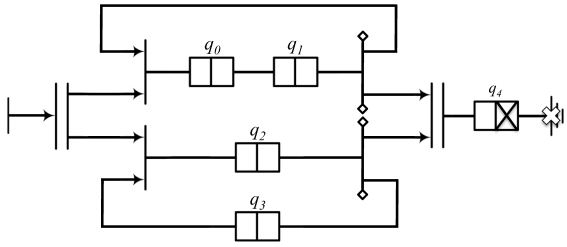


Figure 9.14: *False deadlock found by our tool. The sink is dead and the source injects just one token.*

Deadlocks with false ordering of packets Our approach finds false deadlocks in networks where deadlock is prevented by a specific ordering of messages. Consider the network in Figure 9.15. Blue and red tokens are injected simultaneously into queues q_0 and q_1. The contents of these queues are always equal, including the order in which the tokens are queued. Our algorithm returns a false deadlock, where queue q_0 contains – in this order – a blue token and a red token, and q_1 a red token followed by a blue one. Invariants dealing with the order of different packets in different queues are needed to exclude this false deadlock.

Deadlocks due to rational variables Consider the network in Figure 9.16. One packet is injected at the source. This packet is randomly send to either queue q_0 or queue q_1. As only one packet can reach the join, the join is

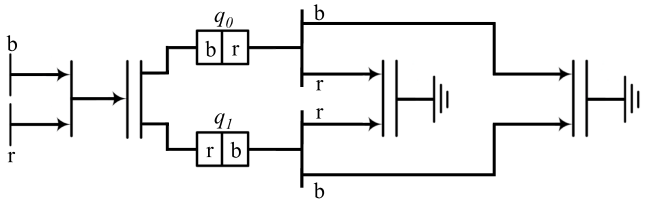


Figure 9.15: *False deadlock found by our tool.*

dead. Consequently, queues q_2, q_3, q_4 and q_5 will always be empty. Our tool returns this deadlock. However, a false deadlock is returned as well. During generation of invariants the variables are not considered integers but rationals, because of efficiency reasons. The invariant generator first assesses that $\#q_0 + \#q_1 \leq 1$. Subsequently, it states that queue q_2 can have at most $^1/_2$ packets. As variable $\#q_2$ is considered an integer by our algorithm, the algorithm correctly derives that queue q_2 is always empty. However, the invariant generation duplicates the half packet to queues q_3 and q_4. Thus the invariant $\#q_3 + \#q_4 \leq 1$ is found. The invariant generation proceeds with adding the invariant that $\#q_5 \leq 1$, as queue q_5 is the sum of queues q_3 and q_4. Based on this invariant, our algorithm finds a false deadlock where queue q_5 contains a packet. The issue is, intuitively, that $(1/2) \cdot 2$ is equal to 1 for rational numbers, but is equal to 0 for integers.

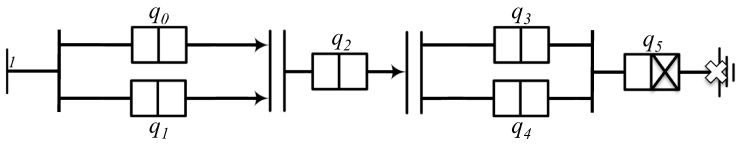


Figure 9.16: *False deadlock found by our tool. The crossed sink is dead and the source injects just one token.*

9.3 Correctness Proof

The algorithm presented in the previous section heavily relies on a translation from semantics to blocking and idle formulas. We present these translations and prove theorems stating their correctness. Using these theorems, we prove correctness of the algorithm for *any* language L in the family of languages presented in Section 9.1.

9.3.1 Automatic Generation of Blocking and Idle Formulas

The blocking and idle formulas of a primitive is expressed using the syntax specified in Figure 9.17. Formulas may consist of linear equations, predicates **Block** and **Idle**, auxiliary functions, and the Boolean operators `and`, `or` and `not`. Signal

`i.select` can be used to assume that channel i is selected. Formulas will be written in `typewriter` font.

$$
\begin{array}{lcl}
\mathsf{form} & := & (\mathsf{form}\ \texttt{and}\ \mathsf{form}) \mid (\mathsf{form}\ \texttt{or}\ \mathsf{form}) \mid \texttt{not}\ \mathsf{form} \\
& & \mid \texttt{forall}\ \mathit{name} \cdot \mathsf{form} \mid \texttt{exists}\ \mathit{name} \cdot \mathsf{form} \\
& & \mid \texttt{true} \mid \texttt{false} \mid \texttt{lin. eq.} \mid \mathbf{Block}(\mathit{name}, \mathit{name}) \mid \mathbf{Idle}(\mathit{name}, \mathit{name}) \\
& & \mid \texttt{i.select} \implies \mathsf{form}
\end{array}
$$

Figure 9.17: *Syntax in which blocking and idle formulas can be defined*

Given the behaviour of some primitive x specified in the syntax presented in Figure 9.5, we automatically generate blocking formulas according to the rules given in Figure 9.18.

Let S be some primitive semantics. The translation of S is parameterized with the current packet p. The translation of the primitive semantics S into blocking formulas for packet p is denoted by $[\![S]\!]^p_{\mathbf{B}}$. Intuitively, the translation of the semantics of a signal yields the conditions under which the signal can always eventually be set to true. We translate signal $i_n.trdy$, which yields the conditions under which this signal can always eventually be set to true. The negation of these conditions represent the conditions under which $i_n.trdy$ can never be set, i.e., the conditions under which channel i_n is eventually permanently blocked.

$$
\begin{array}{rcl}
[\![\mathit{signal}]\!]^p_{\mathbf{B}} & := & [\![\mathit{semantics}]\!]^p_{\mathbf{B}} \text{ iff } \mathit{signal} ::= \mathit{semantics} \in S \\
[\![o_m.trdy]\!]^p_{\mathbf{B}} & := & \texttt{not}\ \mathbf{Block}(o_m) \\
[\![i_n.irdy]\!]^p_{\mathbf{B}} & := & \texttt{true} \\
[\![i_{n'}.irdy]\!]^p_{\mathbf{B}} & := & \texttt{not}\ \mathbf{Idle}(i_{n'}, \texttt{ANY}(\texttt{i}_{\texttt{n}'})) \\
[\![i_n.data]\!]^p_{\mathbf{B}} & := & p \\
[\![i_{n'}.data]\!]^p_{\mathbf{B}} & := & \texttt{ANY}(\texttt{i}_{\texttt{n}'}) \\
[\![f(S)]\!]^p_{\mathbf{B}} & := & \texttt{f}([\![S]\!]^p_{\mathbf{B}}) \\
[\![(S_0\ \mathit{and}\ S_1)]\!]^p_{\mathbf{B}} & := & ([\![S_0]\!]^p_{\mathbf{B}}\ \texttt{and}\ [\![S_1]\!]^p_{\mathbf{B}}) \\
[\![(S_0\ \mathit{or}\ S_1)]\!]^p_{\mathbf{B}} & := & ([\![S_0]\!]^p_{\mathbf{B}}\ \texttt{or}\ [\![S_1]\!]^p_{\mathbf{B}}) \\
[\![\mathit{not}\ S]\!]^p_{\mathbf{B}} & := & \texttt{not}[\![S]\!]^p_{\mathbf{B}} \\
[\![\mathit{forall}\ S]\!]^p_{\mathbf{B}} & := & \texttt{forall}[\![S]\!]^p_{\mathbf{B}} \\
[\![\mathit{exists}\ S]\!]^p_{\mathbf{B}} & := & \texttt{exists}[\![S]\!]^p_{\mathbf{B}} \\
[\![i.select]\!]^p_{\mathbf{B}} & := & \texttt{forall i}' \neq \texttt{i} \cdot \texttt{i}'\texttt{.select} \implies [\![i'.trdy]\!]^{\texttt{ANY(i}')}_{\mathbf{B}}
\end{array}
$$

Figure 9.18: *Translation rules from primitive semantics to blocking formulas.*

Signal $o_m.trdy$ is eventually set if and only if channel o_m can never be permanently blocked. Signal $o_m.trdy$ is translated to the negation of the blocking

formulas of the output channel. For signal $i_{n'}.trdy$ there are two cases. If $n' = n$, trivially the signal is eventually set, as we assume channel i_n contains a packet and thus the initiator of this channel is set. Otherwise, the channel is eventually set if it is not permanently idle for some packet p'. Constant `ANY(c)` indicates that the current packet can be any packet that can be in channel c. The translation of data signal $i_{n'}.data$ of an input channel requires the same case distinction. If $n' = n$, the packet in channel i is simply p, which is a parameter to the translation. Otherwise, the channel contains any type of data. Finally, signal $i.select$ is eventually true if and only if any other in-channel i' can never be permanently selected. When translating $i.select$, the translation will remember that channel i is selected. This is used to translate the reserved channel name i_{sel} with the translation of i. Also, this is required to ensure termination of the translation.

Running Example 9.5 The primitive semantics of the switch are translated as follows:

$$
\begin{aligned}
\mathbf{Block}(i) \quad &\equiv \quad \texttt{not}([\![i.trdy]\!]^p_{\mathbf{B}}) \\
&\equiv \quad \texttt{not}(\ ([\![i.irdy]\!]^p_{\mathbf{B}} \texttt{ and } [\![s(p)]\!]^p_{\mathbf{B}} \texttt{ and } [\![o_1.trdy]\!]^p_{\mathbf{B}}) \texttt{ or} \\
&\qquad\quad ([\![i.irdy]\!]^p_{\mathbf{B}} \texttt{ and not} [\![s(p)]\!]^p_{\mathbf{B}} \texttt{ and } [\![o_2.trdy]\!]^p_{\mathbf{B}})) \\
&\equiv \quad \texttt{not}(\ (\texttt{true and } s(p) \texttt{ and not } \mathbf{Block}(o_1)) \texttt{ or} \\
&\qquad\quad (\texttt{true and not } s(p) \texttt{ and not } \mathbf{Block}(o_2))) \\
&\equiv \quad (\texttt{not } s(p) \texttt{ or } \mathbf{Block}(o_1)) \texttt{ and } (s(p) \texttt{ or } \mathbf{Block}(o_2))
\end{aligned}
$$

We define the translation of primitive semantics to idle equations (see Figure 9.19). This translation is dual to the translation to blocking equations. A notable difference is the translation of data signals at input channels. Say we want to establish idle equations for packet p for output channel o_m. We translate the data at some input channel i_n with the set of all packets p' for which, if signal $o.data$ is translated under assumption $i.data = p'$, this translation yields packet p.

9.3.2 Correctness Proofs of Translations

The proofs in this section uses equalities of LTL formulas as they are defined by Baier and Katoen [5]. In addition, we use the following tautologies:

Name	Abbr.	Law
Distributivity of $\Box$	D$\Box$	$\Box(a \wedge b) \equiv \Box a \wedge \Box b$
Distributivity of $\Diamond$	D$\Diamond$	$\Diamond(a \vee b) \equiv \Diamond a \vee \Diamond b$
Partial distributivity of $\Box$	PD$\Box$	$\Box a \vee \Box b \implies \Box(a \vee b)$
Partial distributivity of $\Diamond$	PD$\Diamond$	$\Diamond(a \wedge b) \implies \Diamond a \wedge \Diamond b$
Lemma 9.1	L9.1	$\Box(\Diamond(a) \vee \Diamond(b)) \implies \Box\Diamond(a) \vee \Box\Diamond(b)$
Lemma 9.2	L9.2	$\Diamond(a) \wedge \Diamond(b) \implies \Diamond(a \wedge b)$

We introduce the following adhoc notation: $F@t$ returns true if and only if LTL-formula F is true at time slot t. E.g., $\Box F$ can be expressed as $\forall t \cdot F@t$ (similarly for $\Diamond$). Lemma 9.1 is a general LTL tautology. Lemma 9.2 does not hold for LTL formulas in general, but is specific to xMAS networks. It uses the fact that xMAS networks are persistent. We first prove these lemma's.

$$
\begin{aligned}
[\![signal]\!]^p_{\mathbf{I}} &:= [\![semantics]\!]^p_{\mathbf{I}} \text{ iff } signal \coloneqq semantics \in S \\
[\![i_n.irdy]\!]^p_{\mathbf{I}} &:= \texttt{not}\,\mathbf{Idle}(i_n, [\![i_n.data]\!]^p_{\mathbf{I}}) \\
[\![o_m.trdy]\!]^p_{\mathbf{I}} &:= \texttt{true} \\
[\![o_{m'}.trdy]\!]^p_{\mathbf{I}} &:= \texttt{not}\,\mathbf{Block}(i_{m'}) \\
[\![i_n.data]\!]^p_{\mathbf{I}} &:= \{\mathtt{p}' \mid [\![o.data]\!]^{p'}_{\mathbf{B}} = p\} \\
[\![f(S)]\!]^p_{\mathbf{I}} &:= \mathtt{f}([\![S]\!]^p_{\mathbf{I}}) \\
[\![(S_0 \textit{ and } S_1)]\!]^p_{\mathbf{I}} &:= ([\![S_0]\!]^p_{\mathbf{I}} \texttt{ and } [\![S_1]\!]^p_{\mathbf{I}}) \\
[\![(S_0 \textit{ or } S_1)]\!]^p_{\mathbf{I}} &:= ([\![S_0]\!]^p_{\mathbf{I}} \texttt{ or } [\![S_1]\!]^p_{\mathbf{I}}) \\
[\![not\ S]\!]^p_{\mathbf{I}} &:= \texttt{not}[\![S]\!]^p_{\mathbf{I}} \\
[\![forall\ S]\!]^p_{\mathbf{I}} &:= \texttt{forall}[\![S]\!]^p_{\mathbf{I}} \\
[\![exists\ S]\!]^p_{\mathbf{I}} &:= \texttt{exists}[\![S]\!]^p_{\mathbf{I}} \\
[\![i.select]\!]^p_{\mathbf{I}} &:= \texttt{forall i}' \neq \texttt{i} \cdot \texttt{i}'\texttt{.select} \implies [\![i'.trdy]\!]^{i'.data}_{\mathbf{I}}
\end{aligned}
$$

Figure 9.19: *Translation rules from primitive semantics to idle formulas.*

Lemma 9.1

$$\Box(\Diamond(S_0) \vee \Diamond(S_1)) \quad \Longrightarrow \quad \Box\Diamond(S_0) \vee \Box\Diamond(S_1)$$

Proof.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\cancel{\neg\Box\Diamond S_0}}{\Diamond\Box\neg S_0}}{(\Box\neg S_0)@w}\,\exists w \qquad \dfrac{\dfrac{\Box(\Diamond(S_0)\vee\Diamond(S_1))}{(\Diamond S_0 \vee \Diamond S_1)@t}\,\forall t}{(\Diamond S_0)@t \vee (\Diamond S_1)@t}\,\forall t}{(\Diamond S_1)@t}\,\forall t \geq w}{\Box\Diamond S_1}}{\neg\Box\Diamond S_0 \implies \Box\Diamond S_1}}{\Box\Diamond S_0 \vee \Box\Diamond S_1}
$$

□

Lemma 9.2 Assuming that both semantics S_0 and S_1 must hold for a transfer to occur, if both semantics hold eventually, then eventually both semantics hold.

$$\Diamond(S_0) \wedge \Diamond(S_1) \implies \Diamond(S_0 \wedge S_1)$$

Proof. By assumption we know that there exists n_0 and n_1 such that $S_0@n_0$ and $S_1@n_1$. In [65] the execution semantics of xMAS have been proven persistent over *irdy* and *trdy* signals. This means that these signals do not change from true to false until a transfer occurs. Assume $n_0 < n_1$. By persistency, semantics S_0 will hold at all time slots n such that $n_0 \leq n \leq n_1$ because until time slot n_1 no transfer occurs. In particular, we know that $S_0@n_1$. Similarly we know that $S_1@n_0$ for the

case where $n_1 < n_0$. Thus $(S_0 \wedge S_1)@\max(n_0, n_1)$. As we have a witness at which both S_0 and S_1 hold, we have established $\Diamond(S_0 \wedge S_1)$.

Note that, e.g., $\neg i.irdy$ is not persistent, i.e., even if no transfer occurs, $i.irdy$ may change from false to true. Restriction 5 ensures that S_0 and S_1 do not contain such signals. □

We now prove the major lemma's required for proving correctness of our translations. Lemma 9.3 states the translation of semantics S to blocking formulas yields a formula that is true if and only if the semantics always eventually hold. Lemma 9.4 states the same for the translation to idle formulas. Our final theorems are directly implied by these lemma's.

Lemma 9.3 The translation of semantics S to blocking formulas yields a formula which is true if and only if the semantics are always eventually true.

$$[\![S]\!]^p_{\mathbf{B}} \iff \Box\Diamond S$$

Proof. The proof is by induction on the structure of S. We present some of the interesting cases:

$$\begin{array}{lcll}
[\![o_m.trdy]\!]^p_{\mathbf{B}} & = & \neg\,\mathbf{Block}(o_m.trdy) & \\
& = & \neg\Diamond\Box\neg o_m.trdy & \\
& \iff & \Box\Diamond o_m.trdy & \text{By duality}
\end{array}$$

$$\begin{array}{lcll}
[\![i.select]\!]^p_{\mathbf{B}} & = & \forall\, i' \neq i \cdot \texttt{i'.select} \implies [\![i'.trdy]\!]^{i'.data}_{\mathbf{B}} & \\
& \iff & \forall\, i' \neq i \cdot \texttt{i'.select} \implies \Box\Diamond i.trdy & \text{By IH}
\end{array}$$

The target of any channel i' will become true if it is selected. The initiator is true, by Restriction 3. So any channel $i' \neq i$ will eventually be ready to transfer once it is selected. By Restriction 2 there will be a finite number of transfers until it gets its turn. Thus we have established $\Box\Diamond i.select$.

$$\begin{array}{lcll}
[\![S_0 and S_1]\!]^p_{\mathbf{B}} & = & [\![S_0]\!]^p_{\mathbf{B}} \wedge [\![S_1]\!]^p_{\mathbf{B}} & \\
& \iff & \Box\Diamond(S_0) \wedge \Box\Diamond(S_1) & \text{By IH} \\
& \iff & \Box(\Diamond(S_0) \wedge \Diamond(S_1)) & \text{By D}\Box \\
& \implies & \Box\Diamond(S_0 \wedge S_1) & \text{By L9.2}
\end{array}$$

Lemma 2 applies here, since by Restriction 4 both S_0 and S_1 are needed for a transfer to occur. Since also:

$$\Box\Diamond(S_0 \wedge S_1) \implies \Box(\Diamond(S_0) \wedge \Diamond(S_1)) \qquad \text{By PD}\Diamond$$

We have established that:

$$[\![S_0 and S_1]\!]^p_{\mathbf{B}} \iff \Box\Diamond(S_0 \wedge S_1)$$

$$\begin{array}{lcll}
[\![S_0 \; or \; S_1]\!]^p_{\mathbf{B}} & = & [\![S_0]\!]^p_{\mathbf{B}} \vee [\![S_1]\!]^p_{\mathbf{B}} & \\
& \iff & \Box\Diamond(S_0) \vee \Box\Diamond(S_1) & \text{By IH} \\
& \implies & \Box(\Diamond(S_0) \vee \Diamond(S_1)) & \text{By PD}\Box \\
& \implies & \Box\Diamond(S_0 \vee S_1) & \text{By D}\Diamond
\end{array}$$

Since also:

$$\begin{array}{lcll}
\Box\Diamond(S_0 \vee S_1) & \implies & \Box(\Diamond(S_0) \vee \Diamond(S_1)) & \text{By D}\Diamond \\
& \implies & \Box\Diamond(S_0) \vee \Box\Diamond(S_1) & \text{By L9.1}
\end{array}$$

We have established that:

$$[\![S_0 \; or \; S_1]\!]^p_{\mathbf{B}} \iff \Box\Diamond(S_0) \vee \Box\Diamond(S_1)$$

□

Lemma 9.4 The translation of semantics S to idle formulas yields the conditions under which the semantics will always eventually be true.

$$[\![S]\!]^p_{\mathbf{B}} \iff \Box\Diamond(S \wedge o_m.data = p)$$

Proof. The proof is by induction on the structure of S. We show here the proof for the interesting case, the translation of signal $i_n.irdy$:

$$\begin{array}{lcl} [\![i_n.irdy]\!]^p_{\mathbf{I}} & = & \neg\,\mathbf{Idle}(i_n, [\![i_n.data]\!]^p_{\mathbf{I}}) \\ & = & \neg\,\mathbf{Idle}(i_n, \{p' \mid [\![o_m.data]\!]^{p'}_{\mathbf{B}} = p\}) \end{array}$$

By Lemma 9.3 we know that $[\![o_m.data]\!]^{p'}_{\mathbf{B}}$ translates correctly to $o_m.data$ under assumption that channel i_n is filled with packet p'.

$$\begin{array}{lcl} & \iff & \neg\,\mathbf{Idle}(i_n, \{p' \mid i_n.data = p' \implies o_m.data = p\}) \\ & = & \exists p' \cdot (i_n.data = p' \implies o_m.data = p) \wedge \neg\,\mathbf{Idle}(i_n, p') \\ & = & \exists p' \cdot (i_n.data = p' \implies o_m.data = p) \wedge \\ & & \quad \Box\Diamond(i_n.irdy \wedge i_n.data = p') \\ & \iff & \Box\Diamond(i_n.irdy \wedge o_m.data = p) \end{array}$$

□

The correctness of our translation from primitive semantics to blocking formulas is expressed by the following theorem. It states that a channel is blocked if and only if its translation is false.

Theorem 9.1 For any input channel i_n of a primitive with semantics S, assuming $i_n.data = p$, the channel is permanently blocked if and only if the translation to blocking formulas yields false.

$$\mathbf{Block}(i_n) \iff \neg([\![i_n.trdy]\!]^{i.data}_{\mathbf{B}})$$

Proof.

$$\begin{array}{lcll} \mathbf{Block}(i_n) & = & \Diamond\Box\neg i_n.trdy & \\ & \iff & \neg\Box\Diamond\, i_n.trdy & \text{By duality} \\ & \iff & \neg([\![i_n.trdy]\!]^p_{\mathbf{B}}) & \text{By L9.3} \end{array}$$

□

A similar theorem is proven for the translation from primitive semantics to idle formulas.

Theorem 9.2 For any output channel o_m of a primitive with semantics S the channel is permanently idle for packet p if and only if the translation to idle formulas yields false.

$$\mathbf{Idle}(o_m, p) \iff \neg([\![o_m.irdy]\!]^p_{\mathbf{I}})$$

Proof.

$$\begin{array}{lcll} \mathbf{Idle}(o_m, p) & = & \Diamond\Box(\neg o_m.irdy \vee o_m.data \neq p) & \\ & \iff & \neg\Box\Diamond(o_m.irdy \wedge o_m.data = p) & \text{By duality} \\ & \iff & \neg([\![o_m.irdy]\!]^p_{\mathbf{I}}) & \text{By L9.4} \end{array}$$

□

We have presented and proven correct the automatic generation of blocking and idle formulas. We now use these translations to prove correctness of our algorithm for any language L.

9.3.3 Correctness Proof of the Algorithm

The proof of correctness is by induction on the language L. The base case is when $L = \mathsf{xMAS}_0$, i.e., when only queues, sinks and sources can be used to model the network N. We then prove that adding a custom primitive preserves correctness.

First, we formalize correctness. Let L be a set of xMAS primitives. Let $\mathcal{N}(L)$ denote the set of valid networks that can be expressed using primitives in language L only. Let $Q(N)$ denote the set of queues in network N.

Definition 9.2 Let L be a set of primitives. A deadlock detection algorithm BlockDetect is *correct for* L, notation **correct**(BlockDetect, L), if and only if for any network N defined in the language L it returns true if and only if there exists a deadlock.

$$\begin{aligned}&\textbf{correct}(\textsc{BlockDetect}, L) \iff \\ &\quad (\forall N \in \mathcal{N}(L) \cdot \forall q \in Q(N) \cdot \\ &\qquad ((\exists p \in \tau(q) \cdot \textsc{BlockDetect}(q.out, p, N, L)) \iff \textbf{Block}(q.out))))\end{aligned}$$

A similar definition of correctness is used for IdleDetect.

Theorem 9.3 The algorithm is correct for xMAS_0.

$$\textbf{correct}(\textsc{BlockDetect}, \mathsf{xMAS}_0) \wedge \textbf{correct}(\textsc{IdleDetect}, \mathsf{xMAS}_0)$$

Proof. We prove that for any channel c and any packet p:

$$\textsc{BlockDetect}(c, p, N, \mathsf{xMAS}_0)) \iff \textbf{Block}(c)$$

The proof is by induction over BlockDetect. The base case is when the target of the channel is a sink. As we assume fair sinks, **Block**(c) is false. The inductive case is when the target is a queue q. Two cases arise: either the queue has already been visited, or not. Consider the first case. An invariant of the algorithm is that for any visited queue, the following constraints have been added to the LIAP that is returned by BlockDetect:

$$\bigvee_{p' \in \tau(q)} \#q.p' \geq 1 \wedge \textsc{BlockDetect}(q.out, p', N, L)$$

We will refer to this constraint with ret_0. The induction hypothesis states that ret_0 is necessary and sufficient for a permanently blocked packet p' at the head of queue q. Since channel c can only be permanently blocked if queue q is full and if there is a permanently blocked packet at the head of the queue, the only extra condition that is to be met is that queue q is full. Line 2 of Algorithm 11 adds this constraint to the returned LIAP. The second case, when the queue has not been visited yet, holds for similar reasons.

We prove that for any channel c:

$$\textsc{IdleDetect}(c, c.data, N, \mathsf{xMAS}_0)) \iff \textbf{Idle}(c, p)$$

The proof is by induction over IdleDetect. The base case is when the target of the channel is a source. As we assume fair sources, **Idle**(c, p) is true if and only if

packet p is not injected at the source. The inductive case is when the target is a queue q. A queue is permanently idle for packet p if and only if it does not contain p and its incoming channel is idle for p, *or* if there can be a permanently blocked packet p' at the head of the queue. Two cases arise: either the queue has already been visited, or not. Consider the first case. An invariant of the algorithm is that for any visited queue, the following constraint has been added to the LIAP that is returned by IdleDetect:

$$\begin{aligned}&(\#q.p = 0 \wedge \textsc{IdleDetect}(q.in, p, N, L))\\ \vee \bigvee_{p' \in \tau(q)} p' \neq p \implies &\#q.p' \geq 1 \wedge \textsc{BlockDetect}(q.out, p', N, L)\end{aligned}$$

No new constraints have to be added to the LIAP as this constraint is necessary and sufficient for permanent idleness of queue q for packet p. The second case, when the queue has not been visited yet, holds for similar reasons. □

Theorem 9.4 Let X be a primitive whose semantics are specified in semantics S. Let L be a set of primitives. If an algorithm is correct for L, the algorithm is correct for L $\cup\{X\}$.

$$\begin{aligned}\textbf{correct}(\textsc{BlockDetect}, L) &\implies \textbf{correct}(\textsc{BlockDetect}, L \cup \{X\})\\ \textbf{correct}(\textsc{IdleDetect}, L) &\implies \textbf{correct}(\textsc{IdleDetect}, L \cup \{X\})\end{aligned}$$

Proof. First, we prove that BlockDetect terminates. The algorithm keeps track of visited queues, to ensure termination in case of cycles of queues. Combinatorial primitives are not marked visited, but as there are no combinatorial cycles between queues, the algorithm will always eventually reach a queue when unfolding blocking and idle formulas of combinatorial primitives. The algorithm terminates for all networks without combinatorial cycles.

Secondly, we prove correctness of BlockDetect by induction. For the inductive case, let i be the in-channel of some primitive x currently under exploration. By induction hypothesis, we know that the algorithm correctly computes blocking equations of the out-channels of primitive x and idle equations for the other in-channels. If x is a primitive of type X, by Theorem 9.1 the algorithm unfolds a necessary and sufficient condition for blocking of in-channel i of primitive x. If x is not a primitive of type X, then it is a primitive in xMAS. The algorithm is correct by assumption **correct**(BlockDetect, xMAS).

The exact similar reasoning holds for IdleDetect. □

Finally, we can formulate our main theorem stating correctness of algorithm for any language L.

Corollary 9.1 The algorithm is correct for any set of primitives L.

$$\begin{aligned}\forall L \cdot \textbf{correct}(\textsc{BlockDetect}, \mathsf{xMAS}_0 \cup L)\\ \forall L \cdot \textbf{correct}(\textsc{IdleDetect}, \mathsf{xMAS}_0 \cup L)\end{aligned}$$

9.4 Experimental Results

Section 9.1.3 presented several examples of communication fabrics. In this section, we present experimental results of running our algorithm on these examples and variations on them.

2D Mesh

We experimented with different layouts of masters and slaves. Curve 2D-MS shows the results where all nodes are both master and slave, i.e., they both inject requests and send out responses upon receiving a request. Curve 2D-MS-LR has been obtained with masters on the left part of the mesh and slaves on the right part. We can prove absence of deadlocks in approximately 1.5 seconds for a 16x16 mesh consisting of 4864 primitives. Masters on even columns and slaves on odd columns yields curve 2D-MS-EO. Deadlocks are found within 2 seconds in a 16x16 mesh. As reference, we have also included results on a 2D mesh without masters and slaves, i.e., a standard mesh with XY routing (curve 2D-XY).

Finally, we have also run our tool on a 2D mesh with adaptive west-first routing. Curve 2D-WF shows that proving meshes deadlock-free with adaptive routing can still be done efficiently. We have proven a 16x16 mesh deadlock-free within 1 second.

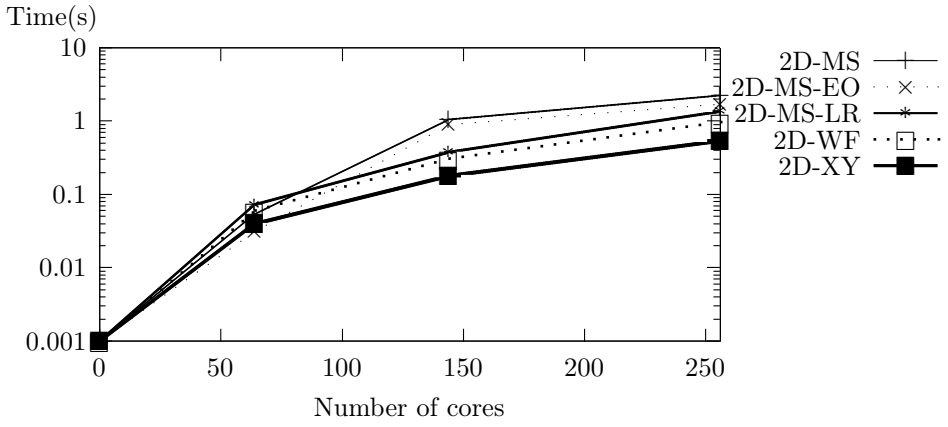


Figure 9.20: *Experimental results*

Spidergon

We have experimented with several variations of the Spidergon chip. First, we have the basic Spidergon with shortest path routing. Our algorithm finds deadlocks instantaneously (curve SP). We have added virtual channels to the ring and modified the routing logic in such a way that deadlocks are prevented. Our algorithm assesses deadlock freedom of a ring with 100 cores (consisting of 1444 xMAS components) in 0,12 seconds.

We have added the write-invalidate cache coherency protocol to the basic Spidergon design, i.e., without virtual channels. This does not increase the running time significantly (curve SP-CCP). Deadlocks are found quickly. We have experimented with limiting the cores that can inject messages. If only the upper right quarter of the ring injects read and write messages, the network becomes deadlock-free. Curve SP-CCP-Q shows the corresponding results: a ring with 32 cores is proven deadlock-free within 30 seconds.

Finally, we have modelled a Spidergon with credit-based flow control (curve SP-CC). We have added a credit control unit to the basic Spidergon, limiting the total number of packets in the ring to some constant C. With each injection of a packet into a local queue, a token is sent to the credit control unit. With each consumption of a packet, a token is removed from the credit control unit. If C is too large, deadlocks are found quickly. With $C = N \cdot k - 1$, where N is the total number of nodes and k the size of the queues, the network becomes deadlock-free. We have established deadlock freedom of a credit-based Spidergon with 32 cores in 193 seconds.

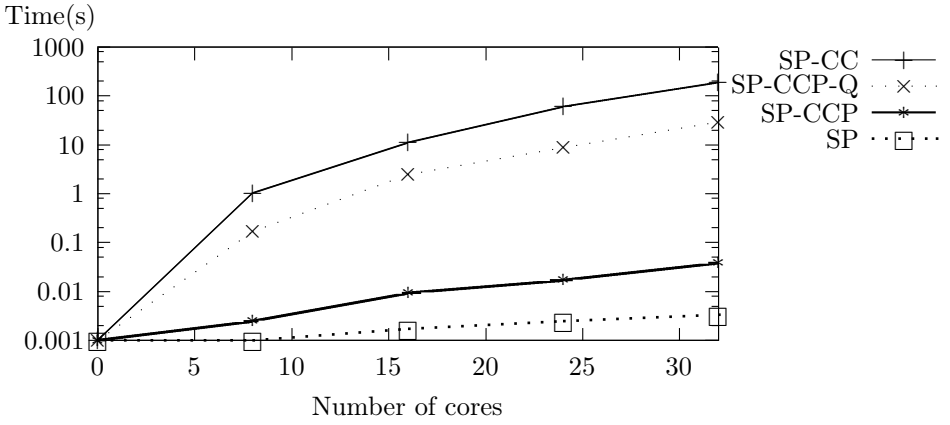


Figure 9.21: *Experimental results*

9.5 Conclusion

This chapter presented an overview of our work on verifying the interconnect integrally. A family of langauges has been presented, together with a deadlock detection algorithm for each language in this family. We have implemented an initial prototype of this algorithm to show the feasibility of our approach.

There is a trade-off between the expressivity of the languages and the tractability of deadlock detection. For example, we restrict the use of the logical negation operator in expressing semantics of new components, to exclude networks where deadlock is dependent on eagerness of channels. Checking efficiently for eagerness is still an open problem. As another example, functions associated to primitives must be unary to ensure that efficient simulation of packet injections

is possible. These restrictions limit the use of some primitives that are commonly used in communication fabrics. Examples are the unrestricted join and the priority merge. However, we were able to model complex examples with, e.g., credit-based flow control, message dependencies, or cache chorency protocols. In all examples, deadlock detection was scalable up to non-trivial sized networks.

The examples presented in this chapter were mostly academic. Applying our algorithm to realistic and industrial examples is very interesting future work. The algorithm heavily relies on invariants to rule out unreachable deadlocks. Currently we have implemented Intels technique for invariant generation. Extending this technique with more accurate and significant invariants would increase the real-world applicability of our algorithm. We also expect it to increase the scalability of our algorithm, as more restrictive invariants generally result in a lesser search space.

In contrast to DCI2, our algorithm requires copious input. It requires a complete microarchitectural model of the application, network and link layer. We are currently working on a graphical tool linked to both the invariant generator and our deadlock detection algorithm to facilitate the design of microarchitectural models. This tool also allows custom primitives, and objects composed of other primitives or other composite objects. The result would enable an incremental and compositional approach to deadlock detection in communication fabrics in the style of D-finder [9, 102].

Epilogue

Summary

This thesis deals with formal and mechanical verification methods for proving correctness of the interconnects on microchips. Future microchips are expected to be built from many cores, which perform computations in parallel. Network-on-Chip (NoC) is a new paradigm in which a sophisticated infrastructure takes care of the communication between the cores.

The ubiquity of microchips mandates precise and thorough methods for establishing their correctness. The contribution of this thesis consists of formal methods focused on the NoC paradigm which are on one hand easy to use and on the other hand scalable. Where there previously was no scalable way to establish of some network that it is always able to successfully complete every pending communication, this thesis presents algorithms which automatically establish this for a large family of networks.

In Part II, we start our effort by formalizing the notion of correctness we are interested in. We coin this notion *productivity*. A network is productive if and only if at all times any message can eventually be injected, and any injected message will always eventually arrive at its destination. Basically, a productive network behaves like a point-to-point out-of-order network with non-deterministic delay.

Productivity is an emergent property depending on interactions between all the constituents of the network. We show that in order to prove productivity of the network as a whole, it suffices to prove several smaller isolated properties on the network constituents. Productivity has been broken down into five network properties: functional correctness, deadlock freedom, livelock freedom, starvation freedom and liveness of injection. Most of these properties have further been broken down into elementary *proof obligations*. Many of these proof obligations are easily discharged. Deadlock freedom, however, is both hard to prove and hard to split into smaller and more elementary proof obligations. Therefore, the remaining parts of this thesis are devoted to automatically proving absence of deadlocks for communication interconnects.

It is common practice to reason over networks in different layers of abstraction using the OSI model. This approach has been applied for deadlock freedom as well. Typically, both the network- and the application layer are proven deadlock-free.

The drawback of this approach is that even when both layers have been proven deadlock-free in isolation, cross-layer deadlocks may still occur due to interactions between layers.

This thesis considers two models of on on-chip communication fabrics. The isolated network model abstracts away from both the application layer and the link layer. Effectively, this isolates the network (that is, the routing, injection, type of switching). Conversely, the integrated network model takes into account all details of the communication fabric, the applications running on top of it and the underlying link layer.

Part III presents both the theory and practice of proving deadlock freedom of communication networks under the isolated network model. We present necessary and sufficient conditions for deadlock freedom in both packet and wormhole networks. For packet networks, we present an algorithm that decides deadlock freedom in $O(|A|)$, with A the number of routing dependencies in the network. We show that a similar polynomial algorithm for wormhole networks is not feasible: deciding deadlock freedom of wormhole networks is shown to be co-NP-complete. We therefore present an algorithm that is incomplete, but polynomial. If it claims that the network is deadlock-free, then this result is sound. However, if a deadlock is found, this deadlock may not actually be legal or reachable from the empty initial configuration. The algorithm decides deadlock freedom in $O(|A||C|)$, with A the number of routing dependencies in the network and C the number of channels.

Subsequently, we incorporate efficient C implementations of our algorithms in the tool DCI2. Besides deadlocks, DCI2 detects livelocks, misrouting and other routing related issues. We have applied DCI2 to various non-trivial examples. We prove correctness of a complicated adaptive fault-tolerant routing function in 20x20 mesh up to two faulty channels. This amounts to proving 2,878,800 configurations deadlock-free. Also, DCI2 confirmed the existence of a deadlock in an initial design of an NoC with irregular routing due to wireless transmissions in the WNoC designed at the University of California, Irvine.

Part IV presents both a language and a deadlock detection algorithm for the integrated network model. The language is based on the xMAS language designed by Intel. The xMAS language is restricted to a set of eight standard primitives, which allows relatively efficient formal verification. We have generalized this language to support user-defined primitives. This generalization ensures that the language is sufficiently expressive, while preserving the possibility of efficient formal verification. Using xMAS with additional user-defined primitives allows the modelling of application layer behavior such as cache coherency protocols and master/slave behavior, the modelling of network layer behavior such as adaptive routing and irregular topologies and the modelling of link layer behavior such as in-network synchronization and counters.

The deadlock detection algorithm presented in Part IV is able to analyze any network that consists of the eight standard xMAS primitives and user-defined primitives. It is therefore able to find cross-layer deadlocks caused by, e.g., message dependencies or wrongly sized counters. The algorithm is not complete. If it establishes deadlock freedom this result is sound, but if it finds a deadlock then this deadlock might not actually be reachable. We have used Intel's invariant

generation technique to rule out unreachable deadlocks as much as possible.

Our algorithm has established deadlock freedom of, e.g., a 2D mesh topology with west-first routing, different types of messages and master/slave application-behavior, or the Spidergon ring topology with a cache coherency protocol and on-chip packet counting.

There is a trade-off between detecting deadlocks of the network in isolation, or applying the integrated approach. The advantages of considering the network layer separately, is that complete – necessary *and* sufficient – conditions can be defined. For packet networks, such a condition is decidable in polynomial time. For wormhole networks, a polynomial algorithm can still prove absence of deadlocks of many networks. In contrast, our deadlock detection algorithm for xMAS is exponential and uses solvers of NP-complete problems such as satisfiability and integer linear programming. We have not been able to formulate a *static* necessary and sufficient condition for the absence of deadlocks. As a result, any deadlock found still has to be scrutinized in order to assess that it is actually reachable.

Conversely, the advantages of considering the network monolithically is the reliability of a positive result. Once a network has been proven deadlock-free – taken into consideration the applications running on top of it and the link layer underneath it – this result is sound. If one proves the network layer deadlock-free in isolation, one is still not assured of the absence of cross-layer deadlocks.

Overall, both approaches have their value. Since our tool DCI2 requires relatively little input and provides copious output, it can be used during the design phase to debug a network. In contrast, once a stable design has been achieved, one can take the effort of formalizing this design in its entirety to prove absence of deadlock once and for all.

The ACL2 theorem prover has been of crucial importance while creating this thesis. We have extensively made use of typical ACL2 features such as its high degree of automation and the executability of the ACL2 code. Also, various non-typical features like the ability to simulate second-order logic or using single-threaded objects to create efficiently executable LISP code have been of great value. Most importantly, there was a large and apposite library available in GeNoC to build upon and to extend. A major disadvantage is the fact that there is a gap between the ACL2 code and its presentation on paper. Due to the fact that the logic of ACL2 is mostly quantifier-free and first-order, the formulation of various theorems and definitions differ from their clean mathematical representations in this thesis.

An example of the added value of proving theorems in ACL2 can be found in the proofs of correctness of our algorithms in Chapter 7. Both algorithms have a post-processing step. This step corrects some undesired behavior of the initial steps. This correction is only required for some specific networks and for some specific traces of the algorithms. We would not have found the necessity of these post-processing steps without mechanically proving their correctness.

Future Work

To fully justify the relevance of the contents of this thesis, some additional claims require further research. In Section 1.1, we argue that analytical methods are necessary to prove correctness of NoCs, since simulation does not scale to future communication-centric SoCs. It would be interesting to compare the algorithms presented in this thesis with state-of-the-art simulation tools. Similarly, we claim that current model checkers do not scale to future NoCs. We have not yet compared our algorithms with state-of-the-art model checkers. Also, our isolated network model can benefit from more justification. To justify assumptions that, e.g., all sources fairly send messages to all others sources, we could prove that without these assumptions, deciding deadlock freedom becomes NP-complete (even for packet networks).

Even though we have proven co-NP-completeness of deciding deadlock freedom of wormhole networks, we are still planning to make a complete decision procedure. Such a procedure could be obtained by linking the output of our polynomial algorithm to an SMT solver to see whether the found deadlock can actually be filled by pairwise disjoint worms. Linking the output to an integer programming solver could have the benefit of finding minimal deadlocks, which can significantly increase the readability of the output.

DCI2 has been used to prove absence of deadlocks of fault-tolerant routing functions. It is run in a brute-force – but parallel – fashion on millions of different configurations where two channels are faulty. For non-trivial networks, we have not been able to scale to three faulty channels. It is an interesting challenge to deal with faulty channels in a more symbolical fashion and to prevent this combinatorial blow-up.

As for our xMAS deadlock detection algorithm, a very interesting direction of future work is dealing with hierarchies of xMAS networks. To facilitate the creation of composite objects, to prove them deadlock-free in separation, and subsequently to reuse this proof in larger xMAS models may significantly increase the scalability of our algorithm. Combined with a graphical xMAS designer tool, this work may lead to a scalable and user-friendly tool for proving deadlock freedom of a large class of communication fabrics.

Appendix A
Datastructures and Notation

A.1 Sets

A set is a – possibly infinite – collection of objects. A set does not contain duplicates and the order in which the objects are stored is irrelevant. That is, one cannot access the first object in a set. A set S from objects $o_0, o_1, \ldots$ is denoted with $S = \{o_0, o_1 \ldots\}$. Membership of object x in set S is denoted with $x \in S$. The empty set is denoted with $\emptyset$. The cardinality of set S is denoted with $|S|$.

Two sets S_1 and S_2 are equal, notation $S_1 = S_2$, if and only if all their objects are equal.

$$S_1 = S_2 \quad \stackrel{\text{def}}{=} \quad (\forall x \in S_1 \cdot x \in S_2) \wedge (\forall x \in S_2 \cdot x \in S_1)$$

Set S_1 is a subset of set S_2, notation $S_1 \subseteq S_2$, if and only if all objects in S_1 are members of S_2:

$$S_1 \subseteq S_2 \quad \stackrel{\text{def}}{=} \quad \forall x \in S_1 \cdot x \in S_2$$

The powerset of set S, notation $\mathcal{P}(S)$, is the set of all subsets of S.

$$\mathcal{P}(S) \quad \stackrel{\text{def}}{=} \quad P \text{ such that } S' \in P \iff S' \subseteq S$$

Given set S and predicate $P : S \mapsto \mathbb{B}$, the set comprehension from P, notation $\{x \in S \mid P(x)\}$, is defined as the set with all objects in S that satisfy P:

$$\{x \in S \mid P(x)\} \quad \stackrel{\text{def}}{=} \quad S' \text{ such that } \forall x \in S' \cdot x \in S \wedge P(x)$$

The removal of object x from S, notation $S - x$, is defined as the set S without object x.

$$S - x \quad \stackrel{\text{def}}{=} \quad S' \text{ such that } x' \in S' \iff x' \in S \wedge x' \neq x$$

Given two sets S_1 and S_2 the union of S_1 and S_2, notation $S_1 \cup S_2$, is defined as the set containing exactly all objects of S_1 and S_2.

$$S_1 \cup S_2 \quad \stackrel{\text{def}}{=} \quad U \text{ such that } x \in U \iff (x \in S_1 \vee x \in S_2)$$

Set	$\{o_0, o_1, \ldots\}$
Membership	$x \in S$
Empty set	$\emptyset$
Cardinality	$\|S\|$
Equality	$S_1 = S_2$
Subset	$S_1 \subseteq S_2$
Powerset	$\mathcal{P}(S)$
Set comprehension	$S' = \{x \in S \mid P(x)\}$
Removal	$S - x$
Union	$S_1 \cup S_2$
Mutual union	$\bigcup S$
Intersection	$S_1 \cap S_2$
Mutual intersection	$\bigcap S$

Table A.1: *Overview of set-related notation.*

Given a set of sets S, the mutual union of S, notation $\bigcup S$ is defined as the union of all objects in S.

$$\bigcup S \quad \stackrel{\text{def}}{=} \quad U \text{ such that } x \in U \iff (\exists S' \in S \cdot x \in S')$$

Given two sets S_1 and S_2 the intersection of S_1 and S_2, notation $S_1 \cap S_2$, is defined as the set containing exactly all shared objects of S_1 and S_2.

$$S_1 \cap S_2 \quad \stackrel{\text{def}}{=} \quad I \text{ such that } x \in I \iff (x \in S_1 \wedge x \in S_2)$$

Given a set of sets S, the mutual intersection of S, notation $\bigcap S$ is defined as the intersection of all objects in S.

$$\bigcap S \quad \stackrel{\text{def}}{=} \quad I \text{ such that } x \in I \iff (\forall S' \in S \cdot x \in S')$$

Table A.1 provides an overview.

A.2 Lists

A list is a – finite – collection of objects. A list may contain duplicates. The order in which the objects are stored is relevant. A list L from objects $o_0, o_1, \ldots, o_k$ is denoted with $L = [o_0, o_1 \ldots, o_k]$. Membership of object x in list L is denoted with $x \in L$. The empty list is denoted with []. The number of objects in list L is denoted with $|L|$. Given a natural number $n < |L|$, the nth object of list L can be accessed by $L[n]$. The head of the list is defined as the first object, i.e., $L[0]$. The tail of list L, notation tail(L), is defined as the remainder.

Two lists L_1 and L_2 are equal, notation $L_1 = L_2$, if and only if they have the same object at each index.

$$L_1 = L_2 \quad \stackrel{\text{def}}{=} \quad |L_1| = |L_2| \wedge \forall 0 \leq n < |L_1| \cdot L_1[n] = L_2[n]$$

The removal of object x from list L, notation $L - x$, is defined as list L without object x.

$$L - x \stackrel{\text{def}}{=} \begin{array}{ll} \text{if } |L| = 0 & [] \\ \text{if } L[0] = x & \text{tail}(L) - x \\ \text{otherwise} & [L[0], \text{tail}(L) - x] \end{array}$$

The last object in list L, notation $\text{last}(L)$, is the object at the end of the list.

$$\text{last}(L) \stackrel{\text{def}}{=} L[|L| - 1]$$

The count of object x in list L, notation $\text{count}(x, L)$, is the number of occurrences of x in L.

$$\text{count}(x, L) \stackrel{\text{def}}{=} \text{count}(x, \text{tail}(L)) + \begin{array}{ll} \text{if } L[0] = x & 1 \\ \text{otherwise} & 0 \end{array}$$

List L_1 is a sublist of list L_2, notation $L_1 \sqsubseteq L_2$, if and only if each object in L_1 has at least as many occurrences in L_2.

$$L_1 \sqsubseteq L_2 \stackrel{\text{def}}{=} \forall x \in L_1 \cdot \text{count}(x, L_1) \leq \text{count}(x, L_2)$$

List L is a subset of set S, notation $L \subseteq S$, if and only if each object in L is a member of S.

$$L \subseteq S \stackrel{\text{def}}{=} \forall x \in L \cdot x \in S$$

The list powerset of set S, notation $\mathcal{L}(S)$, is defined as the set containing any list L that is a subset of set S.

$$\mathcal{L}(S) \stackrel{\text{def}}{=} S' \text{ such that } L \in S' \iff L \subseteq S$$

Given list L and predicate $P : L \mapsto \mathbb{B}$, the list comprehension from P, notation $[x \in L \mid P(x)]$, is defined as the list with all objects in L that satisfy P. In the list, the order of the elements is preserved:

$$[x \in L \mid P(x)] \stackrel{\text{def}}{=} \begin{array}{ll} \text{if } |L| = 0 & [] \\ \text{if } P(L[0]) & [L[0]([x \in \text{tail}(L) \mid P(x)])] \\ \text{otherwise} & [x \in \text{tail}(L) \mid P(x)] \end{array}$$

List L_1 is a permutation of list L_2, notation $L_1 \backsimeq L_2$, if and only if they have the same objects:

$$L_1 \backsimeq L_2 \stackrel{\text{def}}{=} L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1$$

The union of two lists $L_1 = [o_0^1, o_1^1, \ldots, o_k^1]$ and $L_2 = [o_0^2, o_1^2, \ldots, o_m^2]$, notation $L_1 \sqcup L_2$, is the appending of L_2 after L_1.

$$L_1 \sqcup L_2 \stackrel{\text{def}}{=} [o_0^1, o_1^1, \ldots, o_k^1, o_0^2, o_1^2, \ldots, o_m^2]$$

The mutual union of a set of lists S, notation $\bigsqcup S$, is the set containing all objects in all lists in S.

$$\bigsqcup S \quad \stackrel{\text{def}}{=} \quad U \text{ such that } x \in U \iff (\exists L \in S \cdot x \in L)$$

The intersection of two lists L_1 and L_2, notation $L_1 \sqcap L_2$, is a list with all objects in lists L_1 and L_2.

$$L_1 \sqcap L_2 \quad \stackrel{\text{def}}{=} \quad \begin{array}{ll} \text{if } |L_1| = 0 & [] \\ \text{if } L_1[0] \in L_2 \wedge L_1[0] \notin \text{tail}(L_1) & [L_1[0](\text{tail}(L_1) \sqcap L_2)] \\ \text{otherwise} & \text{tail}(L_1) \sqcap L_2 \end{array}$$

The mutual intersection of a set of lists S, notation $\bigsqcap S$, is the set containing all objects shared by all lists in S.

$$\bigsqcap S \quad \stackrel{\text{def}}{=} \quad U \text{ such that } x \in U \iff (\forall L \in S \cdot x \in L)$$

Set of lists S is pairwise disjoint, notation $\bigsqcap S = []$, if and only if each list in S shares no objects with the other lists in S:

$$\bigsqcap S = \emptyset \quad \stackrel{\text{def}}{=} \quad \forall L_1, L_2 \in S \cdot L_1 \neq L_2 \implies L_1 \sqcap L_2 = []$$

Set of lists P is a partition for list L, notation $P \oplus L$, if and only if P is a set of pairwise disjoint lists that cover L.

$$P \oplus L \quad \stackrel{\text{def}}{=} \quad \bigsqcap P = \emptyset \wedge \bigsqcup P \backsimeq L$$

Table A.2 provides an overview.

A.3 Tuples

A tuple is a – finite – collection of objects. Each object can be accessed through a key. Given a set of keys $\{k_0, k_1, \ldots, k_n\}$, a tuple type is denoted with $\langle k_0, k_1, \ldots, k_n \rangle$. Given a tuple t, the object stored under key k can be accessed through $t.k$.

A.4 Graphs

A graph G is defined by a set of vertices V and a function $A : V \mapsto \mathcal{P}(V)$. All graphs in this thesis are directed multigraphs. Function A represents the arcs in the graph and assigns a set of neighbors to each vertex. There is an arc (v_0, v_1) in the graph if and only if $v_1 \in A(v_0)$. In all definitions in this section, we assume a graph G defined by vertex set V and arc function $A : V \mapsto \mathcal{P}(V)$.

The set of *parents* of vertex v, notation $\text{parents}(v, G)$, is the set of vertices of which v is a neighbor.

$$\text{parents}(v, G) \quad \stackrel{\text{def}}{=} \quad \{p \in V \mid v \in A(p)\}$$

List	$[o_0, o_1, \ldots, o_k]$
Membership	$x \in L$
Empty list	$[]$
Size	$\lvert L \rvert$
nth object	$L[n]$
Head	$L[0]$
Tail	$\mathrm{tail}(L)$
Equality	$L_1 = L_2$
Remove	$L - x$
Last element	$\mathrm{last}(L)$
Count	$\mathrm{count}(x, L)$
Sublist	$L_1 \sqsubseteq L_2$
Subset	$L \subseteq S$
List powerset	$\mathcal{L}(L)$
List comprehension	$[x \in L \mid P(x)]$
Permution	$L_1 \backsimeq L_2$
Union	$L_1 \sqcup L_2$
Mutual union	$\bigsqcup S$
Intersection	$L_1 \sqcap L_2$
Mutual intersection	$\bigsqcap S$
Pairwise disjoint	$\bigsqcap S = \emptyset$
Partition	$P \oplus L$

Table A.2: *Overview of list-related notation.*

A list of vertices π is a path, notation $\text{path}(\pi, G)$, if and only if each vertex in π is a neighbor of its successor. The head of the path is its first object.

$$\text{path}(\pi, G) \stackrel{\text{def}}{=} \pi \in \mathcal{L}(V) \wedge |\pi| > 0 \wedge \forall 0 < i < |\pi| - 1 \cdot \pi[i] \in A(\pi[i+1])$$

A list of vertices π is a cycle, notation $\text{cycle}(\pi, G)$, if and only if it is a path where the last object is a neighbor of the head of the path.

$$\text{cycle}(\pi, G) \stackrel{\text{def}}{=} \text{path}(\pi, G) \wedge \text{last}(\pi) \in A(\pi[0])$$

A set of vertices S is a *knot*, notation $\text{knot}(S, G)$, if and only if it is a set of vertices where each vertex has at least one neighbor and all neighbors are included in S.

$$\text{knot}(S, G) \stackrel{\text{def}}{=} S \neq \emptyset \wedge \forall v \in S \cdot v \in V \wedge |A(v)| > 0 \wedge A(v) \subseteq S$$

Table A.3 provides an overview.

Graph	$G = (V, A)$
Vertices	V
Arc function	$A : V \mapsto \mathcal{P}(V)$
Parents	$\text{parents}(v, G)$
Path	$\text{path}(\pi, G)$
Head of path	$\pi[0]$
Cycle	$\text{cycle}(\pi, G)$
Knot	$\text{knot}(S, G)$

Table A.3: *Overview of graph-related notation.*

Appendix B
List of Terms

Abbreviation	Meaning
PS	Packet Switching
WHS	Wormhole Switching
Iso	Isolated Network Layer
Int	Integrated Network Layer

Bibliography

[1] H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004. Cited on page 23.

[2] C. Arbib, G. F. Italiano, and A. Panconesi. Predicting deadlock in store-and-forward networks. In *Foundations of Software Technology and Theoretical Computer Science*, volume 338, pages 123–142, 1988. Cited on page 128.

[3] J. H. Bahn, S. E. Lee, and N. Bagherzadeh. Design of a router for Network-on-Chip. *International Journal of High Performance Systems Architecture*, 1:98–105, 2007. Cited on page 131.

[4] J. H. Bahn, S. E. Lee, and N. Bagherzadeh. On design and analysis of a feasible Network-on-Chip (NoC) architecture. In *Fourth International Conference on Information Technology (ITNG '07)*, pages 1033–1038, April 2007. Cited on pages 131, 134, and 135.

[5] C. Baier and J.-P. Katoen. *Principles of model checking*. The MIT Press, 2008. Cited on pages 6, 10, 63, and 167.

[6] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Handbook of Satisfiability*, volume 185, chapter Satisfiability Modulo Theories (26), pages 825–885. IOS Press, 2009. Cited on pages 6 and 156.

[7] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI '02)*, pages 317–330, London, UK, 2002. Springer-Verlag. Cited on page 66.

[8] L. Benini and G. De Micheli. Networks on Chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78, January 2002. Cited on pages 3 and 17.

[9] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, pages 614–619, 2009. Cited on page 175.

[10] C. Berg and C. Jacobi. Formal Verification of the VAMP Floating Point Unit. In *Proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *LNCS*, pages 325–339, 2001. Cited on page 24.

[11] P. E. Berman, L. Gravano, G. D. Pifarré, and J. L. C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 3–12, New York, NY, USA, 1992. ACM. Cited on page 20.

[12] Y. Bertot. A short presentation of Coq. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLS'08)*, pages 12–16, 2008. Cited on page 24.

[13] W. Bevier, W. Hunt Jr, J S. Moore, and W. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989. Cited on page 24.

[14] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together - formal verification of the VAMP. *STTT*, 8(4-5):411–430, 2006. Cited on page 24.

[15] T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-Chip. *ACM Computing Surveys (CSUR)*, 38(1), June 2006. Cited on pages 6 and 65.

[16] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority: NoC based distributed cache coherency. In *First International Symposium on Networks-on-Chip (NOCS)*, pages 117–126, May 2007. Cited on page 65.

[17] L. Bononi and N. Concer. Simulation and analysis of Network on Chip architectures: ring, Spidergon and 2D mesh. In *Proceedings of the conference on Design, automation and test in Europe (DATE'06)*, pages 154–159, 2006. Cited on page 6.

[18] R. V. Boppana and S. Chalasani. A comparison of adaptive wormhole routing algorithms. *SIGARCH Computer Architecture News*, 21(2):351–360, 1993. Cited on page 20.

[19] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A generic model for formally verifying NoC communication architectures: A case study. In *First International Symposium on Networks-on-Chip (NOCS)*, pages 127–136, May 2007. Cited on page 27.

[20] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. Executable formal specification and validation of NoC communication infrastructures. In *Proceedings of the 21st annual symposium on Integrated circuits and system design (SBCCI'08)*, pages 176–181, 2008. Cited on page 27.

[21] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A formal approach to the verification of Networks on Chip. *EURASIP Journal on Embedded Systems*, 2009. Cited on pages 24, 27, and 55.

[22] S. Chatterjee and M. Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In T. Touili, B. Cook, and P. Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*. Springer, July 2010. Cited on pages 24 and 159.

[23] S. Chatterjee and M. Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Formal Methods in System Design*, 40(2):147–169, 2012. Cited on pages 24 and 159.

[24] S. Chatterjee, M. Kishinevsky, and Ü. Y. Ogras. 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, 2010. Cited on pages 24, 150, and 152.

[25] R. C. Chen. Deadlock prevention in message switched networks. In *Proceedings of the 1974 annual conference - Volume 1*, ACM '74, pages 306–310, New York, NY, USA, 1974. ACM. Cited on pages 17 and 20.

[26] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design*, 36:37–64, 2010. Cited on page 66.

[27] Y.-R. Chen, W.-T. Su, P.-A. Hsiung, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen. Formal modeling and verification for Network-on-Chip. In *International Conference on Green Circuits and Systems (ICGCS,10)*, pages 299–304, June 2010. Cited on page 23.

[28] A. A. Chien and J. H. Kim. Planar-adaptive routing: low-cost adaptive networks for multiprocessors. *Journal of the ACM (JACM)*, 42(1):91–123, 1995. Cited on page 20.

[29] G.-M. Chiu. The odd-even turn model for adaptive routing. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):729–738, July 2000. Cited on page 22.

[30] C. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer Aided Design (FMCAD '04)*, pages 382–398. Springer, 2004. Cited on page 66.

[31] M. Coppola, S. Curaba, M. Grammatikakis, G. Maruccia, and F. Papariello. OCCN: A network-on-chip modeling and simulation framework. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'04)*, pages 174–179, 2004. Cited on page 133.

[32] M. Coppola, M. Grammatikakis, R. Locatelli, G. Mariuccia, and L. Pieralisi. *Design of interconnect processing units Spidergon STNoC*. CRC Press, 2009. Cited on pages 11, 27, and 133.

[33] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. Cited on page 22.

[34] R. Cypher and L. Gravano. Requirements for deadlock-free, adaptive packet routing. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing (PODC '92)*, pages 25–33, New York, NY, USA, 1992. ACM. Cited on page 20.

[35] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4:466–475, 1993. Cited on page 20.

[36] W. J. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, (36), 1987. Cited on pages 8, 10, 18, 20, 23, 62, 71, and 127.

[37] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of Design Automation Conference (DAC)*, pages 684–689, 2001. Cited on pages 17 and 65.

[38] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, 2004. Cited on pages 19 and 20.

[39] P. de Massas and F. Pétrot. Comparison of memory write policies for NoC based multicore cache coherent systems. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'08)*, pages 997–1002, March 2008. Cited on page 65.

[40] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. Cited on page 156.

[41] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of Computer Aided Verification (CAV'00)*, pages 53–68, 2000. Cited on page 66.

[42] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23:257–301, 2003. Cited on page 66.

[43] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 4:1320–1331, 1993. Cited on pages 20 and 21.

[44] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067, October 1995. Cited on pages 8, 18, 20, 21, 22, 71, 93, and 95.

[45] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in cut-through and store-and-forward networks. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):841–1067, August 1996. Cited on pages 21, 22, 82, 85, and 91.

[46] J. Duato. Personal website. `http://www.gap.upv.es/~jduato/`, Accessed on 2012-09-14. Cited on page 21.

[47] J. Duato, O. Lysne, R. Pang, and T. Pinkston. Part I: A theory for deadlock-free dynamic network reconfiguration. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):412–427, May 2005. Cited on page 23.

[48] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks, An Engineering Approach*. Morgan Kaufmann Publishers, 2003. Cited on pages 7, 19, 20, 21, 31, 36, 61, 63, 71, 85, 91, 93, and 99.

[49] B. Dutertre and L. de Moura. The Yices SMT solver. Available at http://yices.csl.sri.com/tool-paper.pdf, 2006. Cited on pages 156 and 163.

[50] E. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin / Heidelberg, 2003. Cited on page 66.

[51] E. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin / Heidelberg, 2003. Cited on page 66.

[52] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious stream productivity. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer Berlin / Heidelberg, 2008. Cited on page 64.

[53] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. In *Proceedings of FCT*, pages 274–287. Springer, 2007. Cited on page 64.

[54] E. Fleury and P. Fraigniaud. A General Theory for Deadlock Avoidance in Wormhole-Routed Networks. *IEEE Transactions on Parallel & Distributed Systems*, 9(7):626–638, July 1998. Cited on pages 20 and 21.

[55] A. Fox. Formal Specification and Verification of ARM6. In D. Basin and B. Wolff, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOLS'03)*, volume 2758 of *LNCS*, pages 24–40, 2003. Cited on page 24.

[56] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. Cited on page 111.

[57] M. Fuechsle, J. A. Miwa, S. Mahapatra, H. Ryu, S. Lee, O. Warschkow, L. C. L. Hollenberg, G. Klimeck, and M. Y. Simmons. A single-atom transistor. *Nature Nanotechnology*, 7:242–246, 2012. Cited on page 3.

[58] S. H. Fuller and L. I. Millett. Computing performance: Game over or next level? *IEEE Computer*, 44(1):31–38, 2011. Cited on page 17.

[59] R. A. Gamboa and M. Kaufmann. Nonstandard analysis in ACL2. *Journal of Automated Reasoning*, 27:323–351, 2001. Cited on page 24.

[60] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock prevention in the Æthereal protocol. *Correct Hardware Design and Verification Methods*, 3725/2005:345–348, 2005. Cited on page 23.

[61] C. J. Glass and L. M. Ni. The turn model for adaptive routing. *Journal of the ACM*, 41(5):874–902, 1994. Cited on pages 20, 22, 53, 132, and 155.

[62] E. Goldberg and Y. Novikov. Berkmin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007. Cited on page 6.

[63] K. Goossens. Formal methods for Networks on Chips. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 188–189, June 2005. Cited on page 24.

[64] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In *VLSI Specification, Verification and Synthesis*, pages 73–128, 1987. Cited on page 24.

[65] A. Gotmanov, S. Chatterjee, and M. Kishinevsky. Verifying deadlock-freedom of communication fabrics. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, volume 6538, pages 214–231. 2011. Cited on pages 32, 156, 159, and 168.

[66] C. Grecu, P. Pande, A. Ivanov, and R. Saleh. BIST for Network-on-Chip interconnect infrastructures. In *Proceedings of the 24th IEEE VLSI Test Symposium*, 2006. Cited on page 135.

[67] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'00)*, pages 250–256. ACM, 2000. Cited on page 17.

[68] A. Hansson, K. Goossens, and A. Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007. Cited on pages 8, 10, 67, and 129.

[69] J. Harrison. Floating-point verification. *Journal of Universal Computer Science*, 13(5):629–638, 2007. Cited on page 24.

[70] J. Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLS'09)*, pages 60–66, 2009. Cited on page 24.

[71] A. Helmy, L. Pierre, and A. Jantsch. Theorem proving techniques for the formal verification of NoC communications with non-minimal adaptive routing. In *IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 221–224, April 2010. Cited on page 27.

[72] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindquist. Network on a chip: An architecture for billion transistor era. In *NORCHIP 2000*, November 2000. Cited on page 17.

[73] M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager. Model checker aided design of a controller for a wafer scanner. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):633–647, October 2006. Cited on page 32.

[74] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart. A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, January 2011. Cited on page 17.

[75] W. Hunt Jr. Mechanical mathematical methods for microprocessor verification. In *Proceedings of Computer Aided Verification (CAV'04)*, pages 523–533, 2004. Cited on page 24.

[76] W. Hunt Jr and S. Swords. Centaur technology media unit verification. In *Proceedings of Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 353–367. 2009. Cited on page 24.

[77] W. Hunt Jr, S. Swords, J. Davis, and A. Slobodova. *Use of Formal Verification at Centaur Technology*, pages 65–88. Springer, 2010. Cited on page 24.

[78] M. D. Ianni. Wormhole deadlock prediction. In *Euro-Par'97 Parallel Processing*, volume 1300, pages 188–195, 1997. Cited on page 128.

[79] J S. Moore. An ACL2 proof of write invalidate cache coherence. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, pages 29–38, London, UK, 1998. Springer-Verlag. Cited on page 66.

[80] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86TM floating-point division program. *IEEE Transactions on Computers*, 47(9):913–926, September 1998. Cited on page 24.

[81] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. Cited on page 6.

[82] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22(5):36–45, 2002. Cited on page 133.

[83] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 83–105, 1972. Cited on page 122.

[84] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, February 2001. Cited on page 24.

[85] M. Kaufmann, P. Manolios, and J S. Moore. ACL2 Computer-Aided Reasoning: An Approach, 2000. Cited on page 24.

[86] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000. Cited on page 24.

[87] G. Klein, R. Huuck, and B. Schlich. Operating system verification. *Journal of Automated Reasoning*, 42(2-4):123–124, 2009. Cited on page 24.

[88] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A Network on Chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 105–112, 2002. Cited on page 7.

[89] S. E. Lee, J. H. Bahn, Y. S. Yang, and N. Bagherzadeh. A generic network interface architecture for a networked processor array (nepa). In *Proceedings of the 21st international conference on Architecture of computing systems (ARCS'08)*, pages 247–260, Berlin, Heidelberg, 2008. Springer-Verlag. Cited on page 131.

[90] O. Lysne, T. M. Pinkston, and J. Duato. Part II: A methodology for developing deadlock-free dynamic network reconfiguration processes. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):428–443, 2005. Cited on page 23.

[91] K. Macdonald, C. Nitta, M. Farrens, and V. Akella. Nocs special section: PDG_GEN: A methodology for fast and accurate simulation of on-chip networks. *IEEE Transactions on Computers*, 99(PrePrints), 2012. Cited on page 6.

[92] T. Marescaux, E. Brockmeyer, and H. Corporaal. The impact of higher communication layers on NoC supported MP-SoCs. In *First International Symposium on Networks-on-Chip (NOCS)*, pages 107–116, May 2007. Cited on page 65.

[93] J. Martinez-Rubio, P. Lopez, and J. Duato. A cost-effective approach to deadlock handling in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):716–729, July 2001. Cited on page 20.

[94] T. Mattson. Many-core applications research using the Intel Single-Chip Cloud computer (SCC). Tutorial at The International Symposium on Networks-on-Chip (NOCS'11), May 2011. Cited on page 17.

[95] K. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer Berlin / Heidelberg, 2001. Cited on page 66.

[96] M. Miédard and S. S. Lumetta. *Network Reliability and Fault Tolerance*. John Wiley & Sons, Inc., 2003. Cited on pages 32 and 65.

[97] J. Misra and K. Chandy. A distributed graph algorithm: knot detection. *ACM Transactions on Programming Languages and Systems*, 4(4):678–686, October 1982. Cited on page 78.

[98] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. Cited on page 17.

[99] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching Networks on Chip. *Integration, the VLSI Journal - Special issue: Networks on chip and reconfigurable fabrics*, 38:69–93, October 2004. Cited on pages 27, 53, and 132.

[100] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001. Cited on page 6.

[101] V. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, July 1990. Cited on page 32.

[102] T.-H. Nguyen. *Constructive Verification for Component-based Systems*. PhD thesis, Université de Grenoble, 2010. Cited on page 175.

[103] L. Ni and P. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26:62–76, 1993. Cited on pages 10 and 132.

[104] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Journal of Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg, 2006. Cited on page 24.

[105] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002. Cited on page 24.

[106] S. Obua and T. Nipkow. Flyspeck II: the basic linear programs. *Annals of Mathematics and Artificial Intelligence*, 56:245–272, 2009. Cited on page 24.

[107] U. Y. Ogras, J. Hu, and R. Marculescu. Key research problems in NoC design: a holistic perspective. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'05)*, pages 69–74. ACM, 2005. Cited on page 18.

[108] J. W. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *Formal Methods in Computer Aided Design (FMCAD '09)*, pages 172–179, 2009. Cited on page 66.

[109] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of the Eleventh International Conference on Automated Deduction (CADE'92)*, volume 607, pages 748–752, June 1992. Cited on page 24.

[110] M. Palesi, R. Holsmark, S. Kumar, and V. Catania. A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS '06)*, pages 142–147, 2006. Cited on page 22.

[111] S. Pandav, K. Slind, and G. Gopalakrishnan. Counterexample guided invariant discovery for parameterized cache coherence verification. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin / Heidelberg, 2005. Cited on page 66.

[112] P. Pande, C. Grecu, A. Ivanov, R. Saleh, and G. De Micheli. Design, synthesis, and test of Networks on Chips. *IEEE Design Test of Computers*, 22(5):404–413, September–October 2005. Cited on page 18.

[113] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. In *Proceedings of the SIGCHI conference on Human Factors in computing systems (CHI'06)*, pages 677–680, 2006. Cited on page 66.

[114] F. Pétrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures. In *9th EuroMicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 53–60, 2006. Cited on page 65.

[115] L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications (ACL2'06)*, pages 1–10, 2006. Cited on page 24.

[116] T. Pinkston. Flexible and efficient routing based on progressive deadlock recovery. *IEEE Transactions on Computers*, 48(7):649–669, July 1999. Cited on page 20.

[117] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the tenth*

annual ACM symposium on Parallel algorithms and architectures (SPAA '98), pages 67–76, New York, NY, USA, 1998. ACM. Cited on page 66.

[118] F. Pong and M. Dubois. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transactions on Parallel and Distributed Systems*, 11:989–1006, September 2000. Cited on page 66.

[119] A. Pullini, F. Angiolini, D. Bertozzi, and L. Benini. Fault tolerance overhead in network-on-chip flow control schemes. In *Proceedings of the 18th symposium on Integrated Circuits and Systems Design*, pages 224–229, September 2005. Cited on page 32.

[120] S. Ray and R. Sumners. Combining theorem proving with model checking through predicate abstraction. *IEEE Design Test of Computers*, 24:132–139, March 2007. Cited on page 66.

[121] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *Computer*, 36(4):60–67, April 2003. Cited on page 6.

[122] A. Roychoudhury, T. Mitra, and S. Karri. Using formal techniques to debug the AMBA System-on-Chip bus protocol. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'03)*, pages 828–833, 2003. Cited on page 23.

[123] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register Transfer Level Specification of the AMD-K7 Floating-Point Multiplication, Division and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. Cited on page 24.

[124] G. Salaün, W. Serwe, Y. Thonnart, and P. Vivet. Formal verification of chp specifications with CADP illustration on an asynchronous Network-on-Chip. In *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07)*, pages 73–82, March 2007. Cited on page 23.

[125] J. Schmaltz. Formal specification and validation of minimal routing algorithms for the 2D mesh. In *Proceedings of the 7th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'07)*, pages 40–49, 2007. Cited on page 27.

[126] J. Schmaltz and D. Borrione. A functional approach to the formal specification of networks on chip. In *Formal Methods in Computer Aided Design (FMCAD '04)*, pages 52–66. 2004. Cited on page 27.

[127] J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 20:241–258, May 2008. Cited on pages 24 and 27.

[128] L. Schwiebert and D. Jayasimha. A necessary and sufficient condition for deadock-free wormhole routing. *Journal of Parallel and Distributed Computing*, 32:103–117, 1996. Cited on pages 20, 21, and 99.

[129] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A Method to Remove Deadlocks in Networks-on-Chips with Wormhole Flow Control. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'10)*, 2010. Cited on page 23.

[130] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the System-on-a-Chip interconnect woes through communication-based design. In *Proceedings of the 38th annual Design Automation Conference (DAC'01)*, pages 667–672. ACM, 2001. Cited on page 7.

[131] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11:633–649, October 1989. Cited on page 64.

[132] F. Silla, M. P. Malumbres, A. Robles, P. López, and J. Duato. Efficient adaptive routing in networks of workstations with irregular topology. In *Proceedings of the First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC '97)*, pages 46–60, London, UK, 1997. Springer-Verlag. Cited on pages 20, 22, and 67.

[133] A. K. Somani and N. H. Vaidya. Understanding fault tolerance and reliability. *IEEE Computer*, 30:45–50, April 1997. Cited on page 32.

[134] W. Stalling. *Operating Systems, Internals and Design Principles*. Pearson Education International, 2009. Cited on page 20.

[135] D. Starobinski, M. Karpovsky, and L. A. Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Transactions on Networking*, 11(3):411–421, June 2003. Cited on page 22.

[136] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990. Cited on page 153.

[137] S. Taktak, J.-L. Desbarbieux, and E. Encrenaz. A tool for automatic detection of deadlock in wormhole networks on chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(1), January 2008. Cited on pages 23 and 129.

[138] S. Taktak, E. Encrenaz, and J.-L. Desbarbieux. A polynomial algorithm to prove deadlock-freeness of wormhole networks. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP'10)*, February 2010. Cited on pages 20, 22, 23, 99, 129, 133, and 144.

[139] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002. Cited on page 7.

[140] S. Tota, M. R. Casu, and L. Macchiarulo. Implementation analysis of NoC: a MPSoC trace-driven approach. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI (GLSVLSI'06)*, pages 204–209. ACM, 2006. Cited on page 18.

[141] G. Tsiligiannis and L. Pierre. A mixed verification strategy tailored for Networks on Chip. In *IEEE/ACM International Symposium on Networks on Chip (NOCS'12)*, pages 161–168, May 2012. Cited on page 27.

[142] R. Ubar and J. Raik. Testing strategies for Networks on Chip. In A. Jantsch and H. Tenhunen, editors, *Networks on Chip*, pages 131–152. 2004. Cited on page 135.

[143] B. Vermeulen, J. Dielissen, K. Goossens, and C. Ciordas. Bringing communication Networks on Chip: Test and verification implications. *IEEE Communications Magazine*, 41:74–81, 2003. Cited on page 18.

[144] C. Wang, W.-H. Hu, and N. Bagherzadeh. A Wireless Network-on-Chip design for multicore platforms. In *19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'11)*, pages 409–416, Februari 2011. Cited on pages 131 and 142.

[145] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual Design Automation Conference (DAC'04)*, pages 681–685. ACM, 2004. Cited on page 18.

[146] P. Wolper. Verification: Dreams and reality. Inaugural lecture of the course "The algorithmic verification of reactive systems", online available at `http://www.montefiore.ulg.ac.be/~pw/cours/francqui.html`, 1998. Cited on page 5.

[147] H. Zantema and M. Raffelsieper. Proving productivity in infinite data structures. In *Proceedings of the International Conference on Rewriting Techniques and Applications*, pages 401–416, 2010. Cited on page 64.

[148] M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society. Cited on page 66.

[149] Z. Zhang, A. Greiner, and S. Taktak. A reconfigurable routing algorithm for a fault-tolerant 2D-mesh Network-on-Chip. In *Proceedings of the 45th ACM/IEEE Design Automation Conference (DAC'08)*, pages 441–446, June 2008. Cited on pages 32 and 144.

Samenvatting

Dit proefschrift gaat over formele verificatie van communicatie infrastructuren aanwezig op microchips. Naar verwachting zullen toekomstige microchips vele verschillende processors bevatten, die parallel berekeningen uitvoeren. Netwerk-op-Chip (NoC) is een nieuw paradigma waarbij de communicatie tussen de processors verzorgd wordt door een geavanceerde infrastructuur.

Het feit dat microchips in ons dagelijks leven overal aanwezig zijn maakt het noodzakelijk dat we precieze en grondige methodes hebben om hun correctheid vast te stellen. De bijdrage van dit proefschrift bestaat uit formele methodes toegespitst op het NoC paradigma die enerzijds makkelijk bruikbaar zijn en anderzijds schaalbaar. Waar er eerder geen schaalbare methode bestond om van een netwerk op een microchip vast te stellen dat het altijd in staat is om iedere gewenste communicatie succesvol uit te voeren, presenteert dit proefschrift algoritmes die dit automatisch kunnen bepalen voor een uitgebreide familie van communicatie netwerken.

In Deel II wordt de notie van correctheid geformaliseerd. We noemen deze eigenschap productiviteit. Een netwerk is productief dan en slechts dan als te allen tijde iedere processor uiteindelijk in staat is een boodschap te versturen en als te allen tijde iedere verstuurde boodschap uiteindelijk aankomt bij de correcte bestemming. In een productief netwerk is iedere boodschap vrij van dodelijke omarming (deadlock), rusteloosheid (livelock) en uithongering (starvation). Een verzameling van aannames wordt geformaliseerd die volstaat om een netwerk productief te bewijzen. De aanname die het lastigst is om van een gegeven netwerk te bewijzen, is de afwezigheid van deadlocks.

Delen III en IV presenteren automatische formele methodes om afwezigheid van deadlocks vast te stellen. Deel III gebruikt een communicatie netwerk model waarin alleen het netwerk relevant wordt beschouwd en de toepassingen die ervan gebruik maken weg worden geabstraheerd (het geïsoleerde netwerk model). Eerst worden twee theorieën gepresenteerd – één voor pakket netwerken en één voor wormgat netwerken – die noodzakelijke en afdoende condities bevatten voor afwezigheid van deadlocks. Voor pakket netwerken presenteren we een polynomiale beslissingsprocedure voor de desbetreffende noodzakelijke en afdoende conditie.

Voor wormgat netwerken bewijzen we eerst dat een dergelijke beslissingsprocedure niet bestaat door het probleem co-NP-compleet te bewijzen. Vervolgens wordt een polynomiale procedure gepresenteerd die een conditie afdoende voor afwezigheid van deadlocks beslist.

De bruikbaarheid van de algoritmes wordt aangetoond met behulp van uitgebreide experimentele resultaten. Om deze resultaten te verkrijgen is de applicatie DCI2 geïmplementeerd (Deadlock Checker In Designs of Communication Interconnects). DCI2 verifieert naast afwezigheid van deadlocks ook andere eigenschappen geformuleerd in de verzameling van aannames uit Deel II. Het is daarmee een applicatie die automatisch productiviteit vast kan stellen van netwerken geformuleerd binnen het geïsoleerde network model. DCI2 wordt gebruikt om productiviteit van een complexe adaptieve route functie vast te stellen in een 20 bij 20 vlak, waarin twee willekeurige communicatie kanalen defect kunnen zijn. Dit komt neer op het vaststellen van afwezigheid van deadlocks in 2,878,800 verschillende configuraties. Ook heeft DCI2 de aanwezigheid van een deadlock bevestigd in een concept ontwerp van een netwerk met onregelmatige routes en draadloze transmissies van de Universiteit van California, Irvine.

Deel IV beschouwt deadlock detectie in het geïntegreerde netwerk model. In dit model wordt niet alleen het netwerk zelf in acht genomen, maar ook de applicaties die er gebruik van maken en de onderliggende implementatie van het netwerk. De uitdaging ligt hier in een taal die enerzijds voldoende expressief is om al deze facetten te modelleren en die anderzijds voldoende toegespitst is om efficiënt formeel verifieerbaar te zijn. De taal xMAS, geïntroduceerd door Intel, bevat een verzameling van acht primitieven waarmee een communicatie netwerk kan worden ontworpen. Deel IV breidt deze taal uit met de mogelijkheid om eigen primitieven toe te voegen. Dit maakt de taal expressief genoeg om allerlei geavanceerde facetten van communicatie netwerken te modelleren zoals protocollen voor het coherent behouden van het tijdelijke geheugen, adaptieve routes, synchronisaties op de chip en boodschap tellers. Vervolgens wordt er een algoritme gepresenteerd dat van een netwerk beschreven in de door de gebruiker gedefinieerde primitieven zoekt naar deadlocks. Het resultaat van deze aanpak is een algoritme dat in staat is complexe boodschap afhankelijke deadlocks te vinden. Het algoritme heeft bijvoorbeeld een twee dimensionale topologie met west-eerst routes, verschillende boodschap types en meester/slaaf applicaties vrij van deadlocks bewezen. Tevens heeft het de Spidergon topologie met krediet boekhouding correct bewezen.

De ACL2 bewijs assistent is cruciaal geweest bij de totstandkoming van dit proefschrift. De verzameling van aannames in Deel II is met behulp van de ACL2 bewijsassistent geformaliseerd. Het bewijs dat deze verzameling afdoende is voor productiviteit is mechanisch tot stand gekomen met behulp van ACL2. Ook correctheid van zowel de theoriën als de algoritmes in Deel III is geverifieerd met behulp van de ACL2 bewijs assistent. Er is uitgebreid gebruik gemaakt van ACL2 specifieke mogelijkheden zoals uitvoerbaarheid van de logica, de hoge mate van automatisering wat betreft het vinden van bewijzen en een uitgebreide bibliotheek waarop gebouwd kon worden.

Curriculum Vitae

Freek Verbeek

17 september 1983:
born in Nijmegen

1995 – 2001:
VWO (secondary school), Stedelijk Gymnasium, Nijmegen

2001 – 2008:
M.Sc., Computer Science, Radboud University, Nijmegen

2008 – 2012:
Ph.D. student, MBSD/ICIS, Radboud University, Nijmegen

2012:
Postdoctoral researcher, Open University of The Netherlands, Heerlen