Automated Cache Coherence Model Reduction using Abstraction Patterns
Masters Thesis

G. J. Hut
AUTOMATED CACHE COHERENCE MODEL REDUCTION USING ABSTRACTION PATTERNS

MASTERS THESIS

by

G. J. Hut

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Wednesday, June 1, 2016 at 13:00 PM.

Student number: 834.078.219
Course code: T75317
Thesis committee: Prof. dr. M. C. J. D. van Eekelen (chairman), Open University
Dr. F. Verbeek (supervisor), Open University

An electronic version of this thesis is available at http://dspace.ou.nl/.
Writing this thesis would not have been possible without the support of several people whom I want to gratefully thank.

I wish to thank my supervisor dr. Freek Verbeek for providing me with the opportunity to write this thesis under his guidance. Cache coherence and bisimulation are not simple subjects, and he provided me with enough railing alongside the abyss of complexity I sometimes felt I was in danger of falling into.

I would like to thank my parents and my brothers. They always supported me to pursue my ideals and dreams, and where would I be without the constant positive competition with my brothers.

Finally, without the never ending support of my wife Marjolijn I would not have been able to continue this effort for almost four years, alongside my regular job. I will try to support her the same way back with the study she is now starting.

Geert-Jan Hut
Vleuten, May 2016
## CONTENTS

List of Figures ........................................ v
List of Tables ......................................... vi
Code Listings .......................................... vii
Summary ............................................... viii

1 Introduction ........................................ 1
   1.1 Motivation ........................................ 2
   1.2 Document Structure .............................. 2

2 Research Context .................................... 3
   2.1 Caches ........................................... 3
   2.2 Multi Core ........................................ 3
   2.3 Memory Consistency ............................. 4
   2.4 Cache Coherence ................................. 4
   2.5 Core connectivity ............................... 6
   2.6 The gem5 simulator ............................ 7

3 Research Design ..................................... 8
   3.1 Research subject ................................. 8
   3.2 Research Questions .............................. 8
   3.3 Validation ....................................... 9
   3.4 Deliverables ..................................... 9

4 Related Works ....................................... 10
   4.1 Cache Coherence ................................. 10
   4.2 Networks on Chips .............................. 11
   4.3 Model Checking and Bisimulation .......... 11

5 State Machine Minimization Approach ........ 13
   5.1 Overview ........................................ 13
   5.1.1 network-impacting minimizations ........ 14
   5.2 The minimization process .................... 15
   5.2.1 Implementation ............................. 15
   5.3 Contribution and research boundaries ...... 15

6 State Machine Abstraction Patterns .......... 17
   6.1 Cache coherence components ................. 18
   6.1.1 Coherence devices ........................... 18
   6.1.2 Coherence features ......................... 24
6.2 Abstraction patterns ................................................. 25
  6.2.1 Removal of non-essential actions pattern .................. 26
  6.2.2 Merge similar actions pattern ................................. 27
  6.2.3 Removal of non-essential device pattern .................... 28
6.3 Have we found all abstraction patterns? ......................... 31

7 State Machine Reduction Algorithms .............................. 32
  7.1 State machine equivalence ...................................... 33
    7.1.1 Trace Equivalence ........................................... 33
    7.1.2 Structural equivalence ..................................... 34
    7.1.3 (Bi-)simulation equivalence ................................ 34
    7.1.4 equality between trace equivalence and (bi-)similarity .... 36
  7.2 NFA to DFA conversion .......................................... 39
    7.2.1 subset construction ......................................... 39
    7.2.2 tau-a reduction ............................................. 40
    7.2.3 Haskell Implementation ..................................... 40
    7.2.4 Impact selection of initial state ........................... 41
  7.3 Strong bisimulation minimization ................................ 44
    7.3.1 Mark phase ................................................ 44
    7.3.2 Reduce phase ............................................... 46
  7.4 Do we generate the most minimal solution? ..................... 46

8 Experimental Results ................................................. 47
  8.1 The MI protocol ................................................ 47
  8.2 The MESI protocol .............................................. 48

9 Conclusions and future work ....................................... 61
  9.1 Discussion .................................................... 61
  9.2 Answers to Research Questions .................................. 62
  9.3 Future work ................................................... 63

A Appendix Cache Coherence Protocol details ...................... 65
  A.1 MI protocol .................................................. 65
    A.1.1 Cache controller ........................................... 65
    A.1.2 Directory Controller ...................................... 67
    A.1.3 DMA Controller ............................................ 68
    A.1.4 More information ......................................... 69
  A.2 MESI protocol ................................................ 69
    A.2.1 L1 Cache controller ........................................ 70
    A.2.2 L2 Cache controller ....................................... 72
    A.2.3 Directory controller ....................................... 75
    A.2.4 DMA controller ............................................ 76
    A.2.5 More information ......................................... 77

Bibliography .......................................................... 78
  Books .......................................................... 78
  Scientific Articles ................................................ 78
  Conferences ....................................................... 79
  Technical Documentation .......................................... 79
<table>
<thead>
<tr>
<th>Acronyms</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glossary</td>
<td>81</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 The MSI Protocol ........................................... 5
5.1 State Machine Minimization Approach ........................ 14
6.1 Cache Coherence Devices .................................. 19
6.2 MI cache controller node ................................. 20
6.3 MI directory controller node .......................... 22
6.4 MI DMA controller node .................................. 23
6.5 Partially abstracted state machine with DMA transitions .... 29
7.1 Trace equivalent .............................................. 34
7.2 Trace equivalent, but not structural equivalent .............. 34
7.3 Merging of similar follow up states ......................... 35
7.4 equivalent DFAs ........................................... 36
7.5 Initial state dependent NFA to DFA conversion, starting with $P_1$ ... 41
7.6 NFA to DFA conversion without remaining states ........... 42
8.1 Gem 5 MESI component configuration ....................... 48
8.2 Original gem5 MI cache controller state machine .............. 51
8.3 Original gem5 MI directory controller state machine ........ 52
8.4 Original gem5 MI DMA controller state machine ............ 53
8.5 Minimized gem5 MI cache controller state machine ......... 53
8.6 Minimized gem5 MI directory controller state machine .......... 54
8.7 Original gem5 MESI Layer 1 Cache controller state machine .......... 55
8.8 Original gem5 MESI Layer 2 cache controller state machine ........ 56
8.9 Original gem5 MESI directory controller state machine ...... 57
8.10 Minimized gem5 MESI Layer 1 Cache controller state machine .......... 58
8.11 Minimized gem5 MESI Layer 2 cache controller state machine ........ 59
8.12 Minimized gem5 MESI directory controller state machine .... 60
LIST OF TABLES

8.1 MI minimization results ................................................. 47
8.2 MESI minimization results ............................................. 49
A.1 MI_example cache controller actions .............................. 66
A.2 MI_example directory controller actions .......................... 67
A.3 MI_example DMA controller actions ............................... 69
A.4 MESI L1 cache controller actions ................................. 70
A.5 MESI L2 cache controller actions ................................. 73
A.6 MESI Directory controller actions ................................. 75
A.7 MESI L2 cache controller actions ................................. 77
CODE LISTINGS

6.1 Network-impacting actions .................................................. 26
6.2 Actions without network impact ........................................... 26
6.3 Similar actions in Directory controller ................................. 27
6.4 DMA queue processing in Directory node ............................. 30
State machines of cache coherence protocol implementations are made complex because of the many optimization and other behavioral requirements these implementations support. This makes for efficient cache coherence protocols, but complex state machines. When focusing on only parts of these implementations - like when only the network protocol behavior is to be investigated - this is a disadvantage, as complex state machines are harder to analyze and use in formal verification.

This document first provides a description on why cache coherence is needed, followed by a description of an automatable process to algorithmically minimize complex cache coherence state machines. The process is to be used to facilitate the research which uses model checking on cache coherence protocols, like the research done in [VS12a] and [VS12b] on Network on a Chip deadlock detection. The process consists of a number of abstraction patterns which remove uninteresting coherence state details, followed by a reduction step which which minimizes the abstracted state machine using a bisimulation reduction algorithm. A software tool is provided that implements this process to reduce gem5 simulator (gem5) cache coherence state diagrams. The patterns and process are validated by reducing gem5 state machines using the software tool.

The developed software and the data sets used in the thesis research will be made available to the research community to enable validation and extension of the work.

Keywords: Memory Consistency, Cache Coherence, Network on Chip, State Machine minimization, Reduction algorithms, Abstraction patterns, gem5, bisimulation
The last years have seen a proliferation of the amount of execution cores microprocessors are provided with. Where only two decades ago only high end systems were equipped with multiple cores, now even low-end smart phones or tablets utilize dual core, quad core or even eight core processors. This trend is continuing. As an example, Reilly describes in [Rei+08] a high end configuration where a system was built from 972 processors with each 6 cores. Fang et al. describe in [Fan+14] their experiences with the Intel Xeon Phi, a commercial currently available 60 core microprocessor.

One of the main challenges with multi-core processors is that the applications that run on them must be able to depend on memory to act consistently. So when a thread updates a value in location A, and next sets a flag in location B (maybe to indicate that A was set), there should be methods available to the programmer to make sure that when another thread gets the new value for B, it will then always get the new value for A.

This should even work when locations A and B are on completely separated memory blocks, managed separately by the processor. The rules which programmers must follow and what they must do to enforce the memory behavior they expect are defined with memory consistency models.

Implementations of these memory consistency models are made complex by the caches, queue’s and other components that are added to the processor and its cores to improve performance. These components work on memory blocks or cache-lines, and this requires some way to keep the content of these blocks coherent between the queue’s, caches and main memory. This coherency is enforced by the caches and memory controllers themselves, by way of cache coherence protocols.

Cache coherence protocols are protocols with which cache devices exchange messages to keep each other up to date on the status and content of memory blocks in the system. The messages are exchanged between the cache controllers and directory controllers devices in the system.
1. **Motivation**

The motivation for this research is the realization that even when the cache coherence protocol as well as the used interconnection network are deadlock-free, the interaction between these two layers can still cause deadlocks [VS12b] [VS12a]. To research this, models of the cache coherence protocol and the interconnecting network must be combined so they can be researched as a single model. This single model is fed into model checking software, which can check this complete model for deadlocks. These analysis tools are more effective when the models to analyze are kept simple, as simpler models are less prone to state space explosions.

This paper gives an answer to the question whether the minimization of cache coherence protocols can be automated, so that the number of states that needs to be examined is programatically minimized. Only states and state transitions that are relevant for the network analysis should remain. This state minimization should be done in a way that the model abstraction remains *sound*, so that all network related properties proven on the abstraction remain true on the original model.

The outcome of this research can be used to more efficiently research deadlock related aspects of the combination of these coherence protocol state machines with network models.

1.2. **Document Structure**

This document is setup as follows: in Chapter 2 a number of concepts used in this research are described. This way the context the research is executed in is introduced. Next, Chapter 3 presents the way the research is structured, and states the research questions and deliverables. Chapter 4 presents a number of significant papers and other background material which describe the topic in further detail, and which were studied in the context of this research.

In the next chapters we present the main result of our research. Chapter 5 presents the overall approach on how to minimize state machines. This approach consists of two parts. The abstraction patterns part is further discussed in Chapter 6. The reduction algorithms are described in detail in Chapter 7. Next we provide the results of our experiments in Chapter 8; here we apply the implementation of the process to the device state machines of two gem5 cache coherence protocols. We finish with our conclusions and give pointers to potential future research areas in Chapter 9.
This chapter describes the context, the technical landscape in which the research is done, by highlighting a number of its aspects. The chapter only gives a high level overview of the used concepts and terminology. An excellent and detailed introduction on most of these concepts can be found in “A Primer on Memory Consistency and Cache Coherence” written by Sorin et al. [SHW11].

2.1. CACHES
Since the 90's caches have become commonplace in microprocessors (CPU’s). They are used to limit the impact of the speed difference between a computer’s memory and its processor(s). This speed difference is caused by the fact that processors can request and process data much faster than main memory can deliver. Caches function as an intermediate between the processor and the main memory to speed up this value retrieval, by retaining the retrieved values for a while in the cache. This works because most programs exhibit locality of reference, which means that most programs re-use specific data (and program code) in a short time interval (temporal locality). They tend to also access data in close proximity to an already accessed memory location (spatial locality). The spatial locality aspect is used by fetching not only the used data, but a whole cache-line (normally a 64 or 128 byte block of memory) at once. Additionally the processor can speculatively pre-fetch any cache-lines it expects to be required next. Nowadays these caches are normally placed on-die, so on the same silicon chip as the execution core itself, to minimize the physical distance between the execution unit needing the data and program code and the cache which temporarily holds copies of these values. With the current processor clock speeds this distance has an impact on the time it takes to retrieve data from the caches.

2.2. MULTI CORE
Another way that the performance of modern microprocessors is increased is by placing multiple cores on a single processor die, creating a multi-core processor. The advantage of having multiple cores as compared to a single, very powerful core is that there is a maximum to how fast you can make a single core run. When you want to make a core run faster you need to increase its clock frequency. With an increased frequency you get a much higher
power dissipation heating your processor [Sut05]. So the current answer to the need for increased performance is not to supercharge a single processor core, but to add multiple cores to a processor. Currently, this is one of the preferred ways to make use of the ever increasing amount of transistors that can be placed on a die. However, there is a challenge when you have a system with multiple cores. The applications running on it must obviously be able to use these cores, and use them efficiently. It must support multi-threading, so that an application can run concurrently on more than one core. Only then can an application benefit from the increased performance the multi-core processor brings.

To do this, programmers must be able to make certain assumptions on how the system (the multiple processor cores together with their caches) handle the application's memory accesses and updates, so from a programmers view this memory handling must be dependable and consistent.

2.3. MEMORY CONSISTENCY

As described in the previous section, to allow developers to write correct programs they must be able to depend on how the system deals with memory loads and stores. For instance, if a piece of code in a single thread writes a memory location, and afterwards reads back this location, the value must be the new value. Only if another thread modifies this memory location in the meantime could the value have changed. This becomes an issue when store actions are queued, something that is commonly done as an optimization method [SHW11, p. 37].

Also, to write correct programs the system must provide methods to make sure that when location A is written before location B by one program thread, another program thread can only see the new A value (so not the old value) after reading the new B value. This should even be the case when the A and B locations are on different cache blocks or caches and when instructions are potentially reordered by the processor cores (another optimization method commonly used in modern processors). If the methods to implement this kind of synchronization are not available, threads would not be able to synchronize, and writing correct multi-threaded programs that share memory between threads would become impossible.

Always providing this guarantee is expensive, performance would be impacted. Different memory consistency implementations provide more relaxed levels of optimization. This is done by allowing the programmer or compiler to specify when synchronization is needed. The trade-off is that the programmer or compiler must then do more work to guarantee a correct functioning application, with more room for mistakes.

2.4. CACHE COHERENCE

Cache coherence becomes an issue in multi-core systems, in which each core has its own local cache. The main issue that cache coherence is concerned with is that multiple cores can keep values from a specific memory location cached in their own local- or intermediate caches, effectively creating copies of those values. When one core updates the value for that memory location, other cores must not use the old value anymore, and must not
update other values at memory locations on that cache-line. Instead the other caches must first update their copy of the cache-line or invalidate it by deleting their copy. When a core request a memory location on this cache-line again it will get the new values. This invalidation/update mechanism effectively keeps the caches coherent.

To orchestrate these coherence activities, cache coherence protocols come into play. These coherence protocols are state machines that for each memory caching device determine the state of each cache-line. A well-known cache coherence protocol is for instance the MSI protocol [SHW11, p. 104]. This protocol, shown in Figure 2.1, has three conceptual states in which the cache-line can be, namely: Modified (the core has written in the cache-line, modifying it), Shared (a read but unmodified copy, potentially shared by others) and Invalid (the cache-line contains no valid data).

![Figure 2.1: The MSI Protocol](image)

When a core wants to read a memory location, and its cache does not contain a copy (the cache-line is in invalid state), a GetS message is sent out. This message could request any already cached value before asking the memory controller to obtain the value from main memory.

When a core wants to write to a memory location, and the cache-line is not already in the modified state, a GetM message is sent out. Any other core receiving that message will invalidate their copy of that cache-line by releasing their copy, optionally directly forwarding the modified data to the requesting core.

Apart from the evident overhead needed for the exchange of these messages, additional intermediate states are also needed to implement the message exchanges, for instance because the cache controller needs to wait for results.

More advanced protocols have also been devised. For instance the MESI protocol is an extension to the MSI protocol, extending it with an extra Exclusive state, indicating situations where a core is the only core with a read data copy in its caches [SHW11, p. 115]. When the core then wishes to update this value, no communication is needed with other cores before going to the Modified state.

However, this efficiency increase comes at a cost, as the MESI protocol state machine is much more complex. This makes it more difficult to create an error free implementation.
of the coherence protocol. Typical errors in state machines lead to either the wrong state, but more often to deadlocks. A deadlock means that the state machine encounters a situation (event) it can permanently not handle, so it can permanently not go to a next state and effectively halts forever; a situation an implementation always wants to avoid. However, this is difficult, in [KAC14] Komuravelli investigated the gem5 simulator (gem5) MESI protocol implementation (see Section 2.6) using formal methods and found six errors in the implementation. This was much to the surprise of the original protocol implementers, who thought the implementation to be stable and error-free; given that it was already in use for a number of years. However, Komuravelli also encountered problems verifying the protocol because of its complexity, having to restrict the state diagram to prevent state space explosions. Restricting was done by creating simpler state machines that still exhibited the behavior under test, effectively creating sound abstractions. These abstractions allow the state diagram to become simpler while retaining the aspects of the state diagram that were under scrutiny.

2.5. CORE CONNECTIVITY

To exchange messages, the cache devices need to be connected to one another. With a limited number of devices a shared bus based approach is normally used, where all cache devices are exchanging messages over a single bus. The logic to exchange messages over the bus also provides for message synchronization, as only one message can be sent at a time, and all devices can see all messages. However, this only works up to a certain number of devices, as when the amount becomes larger the single shared bus becomes a limiting factor. This is simply because it cannot keep up with the increased amount of messages exchanged between the different cores.

For systems with larger counts of cores the message bus is normally replaced by a topology of point-to-point connections, or with Network on a Chip (NoC) connectivity. NoC configurations can be ring structures, grids, spidergon meshes (an adaptation of a ring structure where also opposite nodes are linked) or other topologies. Marty described in [Mar08] the impact of different network topologies on cache coherence.

Each core has a router and a number of connections to other cores or devices. Messages are exchanged between these routers using a routing protocol. This routing protocol is responsible for the optimal routing of the messages without creating deadlocks. Deadlocks can occur when a router cannot forward one or more messages to a next destination router because the relevant message queue is full. When a cycle of such queues is created message transmission halts and a deadlock is created, as described in [DS87]. Sorin et al. also dedicate a paragraph to these so called liveness issues [SHW11, p. 186–]. It is the responsibility of the routing protocol to make sure these kinds of cycles never appear. Previous research has provided efficient routing protocols that are proven to be deadlock-free for a certain topology. See for instance [GN92] where Glass and Ni describe a model for deadlock free wormhole routing.

So for cache coherence to work an error free cache coherence protocol needs to be implemented whereby the coherence messages are exchanged using an error free routing protocol for the network configuration. However this is not sufficient. Verbeek and Schmaltz
have shown in [VS12b] that a combination of a coherence protocols and a routing protocol which are in itself deadlock-free still can suffer from deadlocks.

This problem is currently being researched, and one of the issues is that to prove or disprove deadlocks both state machine models must be combined; this creates much more complex models as a result. This requires smaller state machine abstractions to start with to create combined models that can be proven deadlock-free without suffering from state space explosion.

2.6. **The gem5 Simulator**

Having all cores, caches and connectivity on a single die makes it hard to observe what is exactly going on within the processor. For instance measuring timing of detailed instruction executions, and the verification of (coherence) mechanisms used in the processor is hard. It is also difficult to experiment with these concepts in hardware, as creating a die is expensive and time consuming.

Therefore simulators are built to allow engineers to experiment and to better observe what exactly is happening. One of these simulators is the gem5 simulator. This open source hardware simulator provides an implementation of many of the components used in normal PC’s, like CPU’s, memory, IO components, etc. It is a software simulator with the source code publicly available, it allows one to simulate existing hardware, and test new hardware designs. Cache coherence protocol state machines created for gem5 simulator configurations were used as input for this research.
This research has been designed based on the problem description in Section 1.1. The next section describes the research subject, which further details the problem description. Section 3.2 details the research question that is answered during this research. How the research outcome is validated is documented in Section 3.3. The deliverables this research produced are listed in Section 3.4.

3.1. Research Subject

The subject of this research is to determine whether the automatic creation of minimized cache coherence state machines from gem5 simulator (gem5) configurations is possible. The research attempts to programmatically create sound abstractions of these state machines. These are to be simplified so that they still represent the network behavior of the original machine, and - where possible - other non-relevant state machine complexity is removed. In other research these minimized state machines can then be combined with routing logic models, allowing the combination of the coherence state machines and the routing logic to be proven deadlock-free.

The programmatic minimization of the state machines is done by applying abstraction and reduction patterns. These patterns generate proven minimizations for our purpose, they generate sound abstractions of the original state machines which retain their original network routing behavior.

These abstraction and reduction patterns are the main focus of this research and are created by manually inspecting coherence protocol state diagrams, whereby found minimization methods are investigated on how they can be made generic.

3.2. Research Questions

The question that this research is trying to answer is the following:

Is it possible to create sound minimizations of selected gem5 cache coherence state machines by programmatically applying state machine minimization patterns?

Where minimization patterns are defined as transformations on a state machine that reduce the amount of states or state transitions.
This research question can be divided into and amended with a number of sub-questions:

1. what patterns can be found that can be applied to minimize cache coherence state machines creating sound abstractions?
2. which state machine aspects determine the applicability of these minimization patterns?
3. can programmatically implementable strategies be found when to apply which pattern?
4. can limits be found in the application of these reduction patterns?
5. how can these reduction patterns be best documented?
6. do the patterns have an ordering, is the use of certain patterns enabled by the application of other minimization patterns?
7. if the patterns have an ordering, can an optimal ordering be applied to obtain the best result?

For the last question we define the “best result” as the state machine with the least amount of states, and if equal the least amount of state transitions.

3.3. **Validation**

This research is validated by implementing the found patterns and algorithm in a software tool. It is used to minimize gem5 cache coherence protocol state machines for multiple protocols and devices. The original gem5 state machines are provided in JSON format, and the software tool minimizes these state machines, and provides the minimized state machine also in JSON format for further processing and analysis.

The obtained results can be found in Chapter 8.

3.4. **Deliverables**

The deliverables of the research are the following:

- A master thesis describing the research results, containing:
  - The definition of all found patterns and algorithms for cache coherence state machine minimization and how to apply them;
  - A description of cache coherence protocols which were targeted for minimization, both in their original form and their minimized form;
  - A catalog of analyzed gem5 actions used in the analyzed cache coherence protocols;
- A tool for automatically reducing gem5 state machines.
This chapter describes a number of relevant papers and books that have been investigated to get insight into the area where the research has taken place in.

4.1. Cache Coherence
To investigate memory consistency and cache coherence protocols, the primer written by Sorin et al. [SHW11] was studied. This primer gives an introduction on both these subjects, and introduces various levels of memory consistency and coherence protocols. The coherence protocols are described for both snooping protocols and directory protocols.

Archibald and Baer [AB86] provide another, less verbose introduction in cache coherence protocols. They introduce a number of coherence protocols and try to model performance aspects of these protocols using applications written in the Simula language.

Marty provides another introduction in cache coherence protocols in [Mar08], but with a focus on Network on a Chip (NoC) configurations. He argues that cache coherence is needed in these kind of networks. Chapter 1 and 2 of this thesis were studied for this subject.

Martin et al. provide in [MHS12] arguments ‘Why on-chip cache coherence is here to stay’. They state that on-chip (implemented in hardware) cache coherence implementations can scale to to large amount of cores. They give examples of perceived scaling problems in areas of traffic, storage, maintaining inclusion, latency and of energy usage; and give solutions to each of these problems.

Komuravelli et al. on the other hand describe in [KAC14] problems with hardware implemented cache coherence protocols, by performing a study where they tried to formally verify a gem5 simulator (gem5) implementation of a MESI coherence protocol. They found that even after four years of extensive use bugs still could be found in the implementation. They use this result as evidence that hardware coherence protocols are still hard to understand and design. Because of this they propose that hardware-software co-designed protocols might offer a simpler alternative.
4.2. NETWORKS ON CHIPS

A number of papers were studied about NoC’s and routing. Dally and Seitz present in [DS87] a way to create message networks without deadlocks using wormhole routing. They do this by creating virtual channels with which to separate the message paths. They state that “A deadlock-free routing algorithm can be generated for arbitrarily interconnection networks using the concept of virtual channels”. This removes all potential circular dependencies on network level and makes deadlocks impossible. This would imply that the current research with which to verify cache coherence protocols on NoC networks could be seen as an optimization problem, as any cycle that could create deadlocks could also be removed by adding extra hardware for virtual channels. However, Verbeek et al. describe in [Ver+16] a case where cross-layer deadlocks occur when queues are wrongly sized. This happens even when virtual channels are applied and the routing algorithm itself is deadlock free. These deadlocks emerge because of the interaction of the network topology and routing algorithm and the queue sizes. This research found that queues need to be of a minimum size to avoid deadlocks. Networks made of virtual channels only decrease the minimum queue size, deadlocks still occur when the queue’s are sized below these minimum required sizes.

Verbeek and Schmaltz describe in [VS12a] and [VS12b] work done on verification of coherence protocols on NoC networks. These papers are examples of the kind of research where the minimized cache coherence models provided by this research are to be used in. Given the complexity of these papers, only the introductions and first chapters were studied.

Intel® has created a whitepaper [Int09] on their Quickpath Interconnect interprocessor communication method. This interconnect method is implemented in their current processor line, and features point-to-point connectivity using virtual channels per message type. This would be an example of the application of the theory of Dally and Seitz. Also interesting is the fact that Intel uses point-to-point messages with protocols called home- and source-snooping. These are based on MESI type snooping coherence protocol variations, using point to point connectivity between the different on- and off-die caches.

4.3. MODEL CHECKING AND BISIMULATION

Reduction of state machines is a subject that is investigated in the context of Bisimulation equivalence, or Bisimilarity. Bisimulation is a relatively recent research area, which investigates a stronger equivalence relation compared to trace equivalence, but weaker than structural equivalence. An advantage of bisimulation as an equivalence relation is that the equality relation can be checked efficiently.

Bisimulation is explained in detail in Introduction to bisimulation and coinduction, where Sangiorgi gives an introduction in bisimulation theory, both for strong and weak bisimulation[San12]. Baier and Katoen give a comprehensive summary of many subjects related to model checking in [B+08]. They devote a chapter to equality relations and abstractions, where they also give an introduction into Bisimulation. A short introduction to branching bisimulation is also given in [Bas96].
Bisimulation reduction algorithms are defined that allow for the minimization of state machines, while retaining bisimilarity. Linz describes in [Lin11] a strong bisimulation reduction algorithm, without explicitly naming it as such.
This chapter gives an overview on the outcome of the research we have done. It provides a high level view on the process we devised to minimize the cache coherence state machines. The process generates simpler state machines containing only the features that we are interested in, and that are better suited for model checking.

The overall approach to minimize cache coherence state machines is modeled in Figure 5.1. The details for the reduction and abstraction steps are described in full detail in the next two chapters. There we also describe the specific aspects of the cache coherence state machines we used in the algorithms. We end this chapter by detailing what this research added as new work, and what research results were found in other papers.

5.1. Overview
A cache coherence protocol state machine, or 'state machine', consists of states and transitions between these states. Each state specifies the condition a cached memory block - or cache-line - has for a specific cache coherence device. A cache coherence device can be a of a cache controller, a directory controller or a DMA controller type. Each device contains its own implementation of the state machine; the state machines in all devices combined implement the cache coherence protocol.

Multiple devices of a type can have their own state machine implementations, for instance the MESI protocol has L1 and L2 cache controllers, with unique state machines.

In a state machine, a state transition between states is made when an event is received, normally in the form of messages. On a transition zero or more actions are executed, each action implementes part of the features of a coherence device. This way, an implementation feature, like handling requests from the core or maintaining coherence statistics, is implemented as a set of actions executed on various state transitions.

A cache coherence state machine is a single Strongly Connected Component (SCC); this means that all states in the state machine are always accessible from any other state, directly or indirectly.

Because the Cache Coherence protocol never terminates it has no terminal states. Additionally, we define the protocol as having no initial state. Having no initial state could be
seen as counter-intuitive; one would expect the initial state to be the Invalid state, as implementations normally use. However, an explicit initial state is not needed, as explained in Section 7.2.4.

The state machines consist of long-lived states, like the M, S and I states shown in Figure 2.1. In addition to these long-lived states, the state machines also contains transient states, between the long-lived states. These transient states are used to synchronize between the state machines of the cache coherence devices.

As an example, when a cache controller requires access to a cache-line, it will initially first move from the Invalid state to a transient state. Only after receiving the response message will the state machine move from the transient state to the long-lived Shared state. The intermediate transient state is defined here to allow the device to wait for the response from the memory or directory controller. More complex protocols can define state machines with a sequence of multiple transient states between two long-lived states.

5.1.1. NETWORK-IMPACTING MINIMIZATIONS
What we are interested in for this research is to abstract the state machines to only networking impacting operations. This means that we want to remove all cache coherence features from the state machines that do not have impact on the protocol network behaviour. This includes the content of the messages, we are interested in when messages are sent, to which device node, and of which message type.

We are also only interested in the externally visible behaviour of the state machines;
5.2. THE MINIMIZATION PROCESS

The process we devised to minimize the cache coherence protocol state machines consists of two phases. In the first phase we apply a number of abstraction patterns to the state machine. Each pattern deletes or changes part of the state machine so that non-essential features are removed.

The three patterns we found were named as:

- Removal of non-essential actions;
- Remove non-essential devices;
- Merge similar actions.

These patterns work independently from one another, and are described in detail in Chapter 6.

After the abstraction patterns have been applied, we are left with a sparse SCC state machine; a Non-deterministic finite automaton (NFA) with internal $\tau$-transitions for transitions where all actions have been removed. In this sparse state machine opportunities exist to combine states and transitions, or to remove transitions.

The second phase is the reduction phase. In this phase we apply a strong bisimulation reduction algorithm to reduce the abstracted sparse state machine to minimize the amount of states and transitions. Before applying this reduction algorithm we first convert the sparse state machine from a NFA to a Deterministic Finite Automaton (DFA) to allow the algorithm to be used. To retain the SCC state machine attribute, a post processing step is executed to remove any superfluous state introduced during the NFA to DFA conversion, before performing the strong bisimulation reduction step.

5.2.1. IMPLEMENTATION

The first abstraction pattern and the reduction algorithms are implemented in a software tool written in Haskell. This software tool is fed with a cache coherence state machine and a list of non-network-impacting actions. The tool will minimize the state machine by first applying the abstractions, after which it will use the reduction algorithms to generate the minimized state machine.

With this tool the applicability of the minimization process is shown.

5.3. CONTRIBUTION AND RESEARCH BOUNDARIES

The subset construction, $\tau$-elimination and strong bisimulation reduction algorithms are well-known algorithms. We have used the description in [Lin11] of these algorithms to guide our implementation.
The main knowledge added by this research are the following:

- We have created a number of pattern descriptions to guide in the removal of unnecessary state machine features;
- We have devised a post processing step to adapt the well-known subset construction algorithm to state machines without initial states, and proven its correctness;
- We have provided a formal proof that for Deterministic Finite Automata (DFAs) similarity, bi-similarity and trace equivalence coincide;
- We have devised the overall minimization process, using the above knowledge.

The three patterns we documented were found in the investigated MI and MESI state machines. It could be that other cache coherence state machines provide evidence of additional patterns; these three patterns were found during the inspection of the MI protocol state machines, and were subsequently validated on the MESI state machines.

We could not find a formal proof for the equality of bi-similarity and trace-equivalence for LTS DFAs in the literature. The closest we came was a theorem in [B+08, p.579] that stated the equivalence for AP-deterministic state machines. This theorem was not provided with a proof, this was left as an exercise for the reader. The equivalence between these two equalities is essential to allow the strong bisimular reduction to be used in our algorithm; therefore this paper provides the proof for this theorem.

Together the individual proofs in this document and references to proofs in other documents provide the mathematical proof of the correctness of the complete reduction algorithm.

We did not attempt to prove that the combination of these algorithms provided minimal results. Especially the subset construction algorithm can provide additional states. We did not observe this during our experiments, our expectation is that these additional states are removed again during the strong bisimular reduction step.

Weak bisimilarity is an equivalence relation that can be directly applied on state machines with internal $\tau$-transitions. It could be that research in this area provides more applicable reduction algorithms, resulting in simplifications to more compact NFA versions of the state machines. This is left as future research.
When we want to minimize the gem5 simulator (gem5) cache coherence state machines, we want to remove those components, device nodes and structural features - named features in the rest of this chapter - that are not needed for our purposes.

This chapter presents the abstraction patterns used to remove non-essential features from cache coherence protocol implementations. The described abstraction patterns are the ‘Removal of non-essential actions’ pattern, the ‘Merge similar actions’ pattern, and the ‘Removal of non-essential device’ pattern.

For this research we are interested in the networking behaviour of the state machine implementations. In Section 6.1 we will first identify which components and features are defined to implement the protocol. We indicate which opportunities exist to remove the features we do not need to model for our research. We explain why these features are non-essential for our research by describing them in sufficient detail.

In Section 6.2 we will describe the patterns we have identified that can be used to remove the non-essential features from the state machines. The information in Section 6.1 is used with these patterns to abstract away these non-essential features from the state machines.

These patterns are like recipes; when the proper information on the features is provided, an implementation of the pattern algorithm can programmatically remove the non-essential features. The patterns can in principle be re-used by other research, to remove other subsets of state machine features deemed non-essential for their research.

**Definition** Essential feature - A feature is essential if and only if it can influence the deadlock behaviour of the cache coherence protocol implementation subsystems we are interested in. All other features are deemed non-essential.

For our research, a feature is essential if and only if it influences the way messages are transmitted and received over the on-die communication network. Examples of features that are essential are the queues with which the cores send and receive messages. Features which influence the message routing (addressing) are also seen as essential.

**Definition** Network-impacting - A feature is network-impacting if and only if it directly impacts the network behaviour of the protocol. All other features are deemed non-network-impacting.

The abstractions we create are *sound* and *complete*. 
Definition Sound abstraction - An abstraction is sound if all properties on the abstraction are also present in the original model.

Definition Complete abstraction - An abstraction is complete if all relevant network properties from the original model are also present in the abstraction.

However, we do not guarantee that all non-essential features are removed when these patterns are applied. In Chapter 7 we describe methods to reduce the size of the sparse state machine that remains after the application of these abstraction patterns.

6.1. CACHE COHERENCE COMPONENTS

The gem5 implementation of the cache coherence protocols consists of a number of coherence device nodes, which implement structural features. The coherence device nodes are the component nodes that execute the coherence protocol, like the cache controller(s), directory controller(s) and the DMA controller(s).

The structural features are used to implement these devices, and with which the nodes implement services like message reception and transmission, temporary buffers for cache data, message queues and usage statistics, etc.

In gem5, the device node types can be identified by their cache coherence protocol implementation. Each node has its own implementation source file in the gem5 source-code. This sourcefile describes and implements the state machine that provides the device functionality for that node. The features are implemented as action functions in the state machine implementations. In Appendix A more detail information on the gem5 implementation for the investigated coherence protocols is given.

In this paper we will concentrate on the devices and features found in the examined gem5 protocols. Other protocols can have additional features and device node types to implement additional functionality, for instance to support special caches or additional cache-line states, like the Owned or Forwarded states used in more complex protocols.

Before abstracting, the protocols need to be inspected to obtain the information needed to apply the abstraction patterns. Depending on the outcome of this inspection additional non-essential components and features can be identified.

The terminology and device naming used in this paper follows the gem5 protocol implementation, additional detail on this is given in [Sor+02].

This section will provide a high level description of the devices and structural features we encountered, in sufficient detail to determine their network-impacting state. With this information we can determine if the device or feature can be abstracted away, with the patterns described in the following section.

6.1.1. COHERENCE DEVICES

The coherence devices are the components that together implement and execute the coherence protocol. They communicate with messages, sent and received via message queues. Each device will execute their own state machine, implementing the part of the cache coherence protocol that the device is responsible for. The state machines are executed for cache-lines, which are the memory blocks on which the caching is implemented. For each cache-line the state is separately maintained, and this state is used to run a separate in-
stance of the cache coherence protocol. A cache controller can simultaneously have cache-lines in a read-only state, in a modified state, and in the various transitional states at the same time. These cache states and transitions are independent from one another from a coherence protocol view, so we do not have to look at interactions between the state machine instances for different cache-lines. State transitions within devices are normally executed based on messages that are received from other components. This in turn can lead to messages sent to other components. The message recipients can be other coherence devices or they can be external components, like cores and memory subsystems that interact with the cache coherence subsystem.

As a side note, any multi-threaded program that is executed by multiple micro-processor cores creates dependencies between state machine executions of different cache-lines. The application code defines the order in which cores trigger memory loads and stores to memory locations. We ignore any dependency that is created because of the application, as we define these as being not non-essential to our research.

For the device descriptions we have used the MI devices, other cache coherence protocol configurations will have comparable devices.

**COMMUNICATION NETWORK**

The communication network, or *network*, is responsible for the transport of messages between the other coherence devices. It is not a direct part of the cache coherence protocol implementation, and normally assumed to be a lower level feature when looking at cache coherence protocols. For our research it is however an essential component, as it is this device which is to be model checked with the minimized cache coherence protocols for deadlock behaviour. This device is modeled separately, and the model is later joined with
the minimized cache coherence state machines.

Depending on the network infrastructure, it can be composed of one network or a number of sub-networks. The requests and responses can be transmitted over their own network, and the DMA communication can also be done over a separate network.

One reason to separate requests from responses is that the coherence protocol is synchronized on the request messages. For snooping/bus protocols this is especially important, as all cache controllers need to monitor the request network to synchronize amongst themselves.

In the cache coherence protocol definition, these networks are represented by the different queues the cache coherence protocol implementations use to send and receive messages. Depending on the network configuration, networks can combine messages from different queues. Network implementations will normally not split message from one queue over different networks, as the network normally isn’t aware of the message content; if messages are to be sent over different network, the cache coherence devices will use different queues.

**CACHE CONTROLLER NODE**

A first level cache controller - like used in the MI protocol - is attached to a processing core, and implements the interaction between the core, the local cache or caches and the rest of the cache/memory subsystem. Lower level caches like the L2 cache controller in the MESI protocol provide caching functionality between two component layers.

When a processor core needs to access a memory location for a read or write action, it requests the value from the L1 cache controller.

When the requested cache-line is already present in a private data cache, the controller can immediately write the value or return it. When the cache-line is not in the local cache, the controller will send out a request to obtain a read-only or writeable copy of the cache-line. How this request is executed, how the cache-line is subsequently handled and written back to memory, is the primary focus of the cache coherence protocol.

The gem5 cache controller implementations use queues to decouple the interaction between the controller and the core, as well as to decouple the interaction with the other nodes via the communication network. Figure 6.2 shows the queues that are present in the MI cache coherence protocol implementation we examined.

The incoming requests from the core are queued in the Mandatory Queue, the responses are directly returned to the core sequencer, without queueing. The requests in the mandatory queue come from a sequencer, this is the gem5 core component that initiates and handles memory requests.

The MI cache controller has two sets of queues for requests and responses to and from other components. This separation exists because requests and responses can be handled via separate networks, depending on the exact network configuration. One of the reason
these networks can be split is because the ordering of memory requests for the various cores is done on request network messages.

Requests are transmitted and received via the request network. Outgoing requests are when the current core requires a cache-line not already in the private data cache of the controller. Incoming requests are from the directory controller, if another cache controller requires access to cache-lines that the current cache controller uses (‘owns’). In addition to request type and addressing information, a request message can also contain message data in the case of a 'PUT' request.

The response network is used to send and receive responses on the requests. The responses are either ack's or nacks, to (negatively) acknowledge the handling of requests. The response messages can also contain memory data.

In addition to the mandatory queue shown in Figure 6.2, a core can also communicate with the cache controller via an optional queue, if it exists. This queue can also be used by the execution core to send memory requests to the cache controller. The mandatory queue is used by the execution core to send ‘mandatory’ requests, the requests that must be handled by the coherence subsystem, to the cache controller node. An optional queue however, will be used by the core sequencer to initiate optional requests, like pre-fetches of cache-lines needed in the (near) future. The cache-line is then retrieved from the memory subsystem, and stored in the private data cache. The main difference between the optional and mandatory queue handling is that the results of a fetch via the optional queue are not returned to the sequencer. If the cache coherence subsystem is busy with other requests it can ignore the pre-fetch request. It will always be followed by a request via the mandatory queue when the cache-line is actually needed for a load or store action. We could not find evidence for it, but presumably the name mandatory and optional queue are chosen to indicate the status of memory requests on these queues.

We found an optional queue definition in the gem5 MESI protocol, the MI protocol did not define an optional queue.

When an abstraction is created for the network-impacting aspects of this node, the core communication and private data cache interaction can be removed. All actions that use the queues to both the request- and response networks need to remain. These actions generate and receive the requests and responses that are handled via the communications network that we are researching.

Another reason that responses cannot be abstracted away is that they are at minimum needed to retain the synchronisation between the state machines running on the various device nodes. This implies that we cannot do our research using only request messages.

**Directory controller node**

In the gem5 implementations, the directory controller is responsible for the provisioning of cache-lines from memory subsystems. For protocols where the cache controllers do not communicate with one another directly, the directory controller can also maintain the ownership of the cache-lines. For the MI protocol this is indeed the case, for the MESI protocol this function is implemented in the L2 cache controller. It will also make sure that requests for cache-lines already claimed by other cache controllers will be forwarded to that controller.
When looking at Picture 6.3, the queues drawn above the directory controller communicate with the network and are as such essential for our research. There is one incoming request queue from the cache controllers, but a directory controller has two outgoing queues to the cache controllers. The incoming queue contains the received requests to be handled from cache controllers, via the request network. Incoming requests are forwarded when they are for a cache-line that is currently used ('owned') by another core. One outgoing queue is used to forward these forwarded requests over the request network; the other outgoing queue is used for responses for requests the directory controller could handle. The latter are sent over the response network. The responses of the forwarded requests are sent directly between the cache controllers. The directory controller does not initiate requests itself, and also doesn’t directly receive responses from the network. Because of the latter, it doesn’t contain an incoming response queue.

For the MI protocol the interaction with the DMA controller is done over separate queues, separate from the communication with the cache controllers. DMA requests are received from the DMA controller via a separate incoming queue, and responses are returned via a separate DMA response queue.

Depending on the network configuration the DMA controller queue’s are network-impacting and essential for our research, when the DMA messages are transmitted via the same network.

The directory controller directly sends requests to the memory subsystem, but has an incoming queue for memory responses. These are non-essential for our research, and can be removed.

The directory controller is only needed for protocols which are not ‘snooping’. When a snooping protocol is used, the cache controllers synchronize their controller states by snooping on all requests on the request network. This makes it unnecessary to retain the cache-line states in a directory, and no dedicated controller is needed to forward requests. Obviously we still need a controller to handle requests to the memory device. In the MESI protocol the directory controller is used for access to the memory device, and handle DMA requests.

In the literature the directory controller or memory controller also contain a shared ‘Last Level Cache’, or LLC. This cache is used as a shared secondary cache for the cores, and provides an additional optimization for accesses to the memory subsystem. The gem5 implementations of the cache coherence protocols we investigated do not implement this LLC.

A directory controller node is mandatory for all memory subsystems present in the sys-
tem. At least one will be present in each configuration.

**DMA controller node**

A DMA controller is a special kind of controller that requests cache-lines for block oriented IO actions. In some architectures they can also be used for memory block-copy actions, relieving the core of this work. To implement these DMA actions, the DMA controller sequentially claims a set of cache-lines, either for reading or for writing.

As a DMA controller normally only reads or writes a memory location once, it exhibits excellent spatial locality, but no temporal locality. This makes the memory request behaviour more predictable when compared to a cache controller. A DMA controller therefore does not need to have a private data cache. It does not need to hold on to cache-lines for a longer time and does not need to retain access many blocks simultaneously.

When we look at the controller behaviour from a distance, they look very similar to the cache coherence devices connected to the cores. However, in the protocol implementations we investigated they do run a different, much simpler state machine. If one wants to take this different state machine behaviour into account, one or more DMA controllers should be added to the model. For our research we decided to remove the DMA controllers.

When investigating network-impacting behaviour, another criteria is whether the DMA controller communicates via a network which is shared with the cache controller devices. Only if the network is shared can it impact and cause deadlocks in the communication between the directory- and cache controllers. Otherwise the component will never impact this network behaviour, and can be removed.

This last attribute can only be determined by looking at the network configuration itself.

**Microprocessor core and DMA IO devices**

The microprocessor core and DMA IO devices are responsible for the memory requests issued to respectively the cache- and DMA controllers. For this they contain sequencer components that requests cache-lines from the cache controller.

For our model checking purposes the core and IO devices can be abstracted away, as they do not directly impact network interactions.

**Memory**

The memory subsystem contains the system memory. The cache coherence subsystem communicates with the memory subsystem via the directory controller.

For our model checking purposes we are not interested in the content of the memory or the cache-lines we exchange. We can abstract away this subsystem and the interaction with it. Also any lower level cache or Last Level Cache (LLC) can be abstracted away for this reason, if it is present in a protocol implementation.
6.1.2. **Coherence features**

The structural features discussed in this section are attributes of the implementations which enrich the core protocol with functionality. Some of these functions are mandatory for a correct working of the protocol. Others are supplementary, as they provide performance optimizations or data concerning the state machine execution. An example of the latter is statistics gathering, which can provide performance data like cache hit ratio.

All features that are not impacting the deadlock behaviour we are interested in can be abstracted away.

**Message contents**

One of the things that we are not interested in for our research is the memory data in the exchanged messages. This is because the memory content only becomes relevant outside of the cache coherence subsystem, the subsystem itself does not interpret message content. The core, memory and DMA IO devices are of course interested in the memory content, but these we will also exclude from the checked model.

**TBE Buffers**

The Transaction Buffer Entries (TBEs) are records that hold information on the transactions that are currently being executed [Sor+02, p.5]. One of the uses of the TBEs is to make sure that only one transaction is executed per cache-line per coherence device node. The amount of TBEs present in the node limits the amount of transactions that a node can issue simultaneously.

When not looking at the memory content itself, the main impact to the network behaviour this feature has is the limitation on the amount of transactions that can be issued. This limits the amount of messages that the node can wait on. If that is no concern, when for instance this message limit is already implemented with queue sizes, the feature can be removed.

**Queues**

Queues are the primary means with which the gem5 cache coherence device nodes communicate amongst each other and with the subsystems in the environment. They are used to de-couple the various components, and to allow the cache to defer handling of messages when performing transitions. The queues are shared for all messages, so there is for instance only one mandatory queue used for all memory requests. Any message at the top of the queue will block all further messages if not handled, so special handling is needed when messages are to be deferred.

The queues that are needed in our abstracted model are those that communicate with the network and environment devices that we want to model check. The other queues and their interactions can safely be removed, as they will not impact the behaviour we are interested in. This does not mean that no deadlocks can occur in the interaction with these devices, but this is out of scope for our current research.

For instance the mandatory queue can be removed as it is not network-impacting. The model checking will be done directly on the messages initiated from cache controllers to the directory controllers.

**Request recycling**

The queues that the different components use combine requests for all cache-lines. When a cache-line is in a transient state, the state machine is not capable of handling another
request for this cache-line. If a message would stay on the head of the queue this would also block this queue for all other messages, also for all other cache-lines.

To prevent this blocking and to increase the performance, the request can be recycled. This can for instance happen by either placing it back in the queue, or by placing it into a side table. The request is kept here until the associated memory block state machine is capable of handling this request again.¹

For our purposes this request recycling functionality can be removed, as we are not interested in performance aspects.

**STATISTIC GATHERING**

Statistics gathering is done by gem5 to get insight into the performance of the cache coherence subsystem, for instance to measure the cache hit ratio with differing cache sizes.

This feature can be safely removed during abstraction.

**CACHE-LINE OWNERSHIP**

Cache-line ownership is a feature that is present in the directory controllers. It is needed to forward requests for already used (‘owned’) memory blocks to the cache controller node that is currently owning the block. As such, it defines addressing for forwarded messages. This makes it an network-impacting feature.

For the MI protocol the cache-line ownership is tracked by the directory controller. This controller tracks which device has the cache-line in Modified state. For the MESI protocol the L2 cache controller tracks the list of L1 caches that have the cache in Shared state, or the cache that has the cache-line in Exclusive/Modified state.

### 6.2. Abstraction Patterns

The abstraction patterns provide a structured method to allow automated removal of non-essential protocol features. We have documented three patterns, each of these patterns implements a way to remove non-essential functionality to minimize the size of the state machine.

The three documented patterns are named as:

- Removal of non-essential actions;
- Merge similar actions;
- Removal of non-essential device.

These three patterns are described in detail in the following sections. We assume that when actions are deemed non-essential, they can be abstracted away with these three patterns.

Our methodology and these three patterns are *sound* because we don't add features or transitions, and *complete* because all removed features are non-essential.

As mentioned in the introduction of this chapter, we do not guarantee that all non-essential features are removed when these patterns are applied.

¹See [http://www.m5sim.org/SLICC#Special_Functions](http://www.m5sim.org/SLICC#Special_Functions) for more information on the features gem5 provides for request recycling.
This could be caused by the way that the patterns are applied, in that we didn't identify all non-essential features. Also, more complex protocols could also allow identification of additional abstraction patterns, which could allow removal of even more non-essential features.

### 6.2.1. Removal of non-essential actions pattern

The gem5 state machine state transitions are all labelled with an event and a list of actions. Each action in this list is an action that is executed when the event is received, executed during the state transition. Events are received and responses are sent via the queues used for message exchange. As we are only interested in the networking related behaviour, we only need to look at the actions that use networking related queues or directly influence the message sending like the cache ownership feature. The network related queues are the queues that are used to communicate between the components running the cache coherence protocol.

What we will do is inspect for each action in all protocol components whether it performs an enqueue or dequeue of a messages on any of the networking queues. When the action does not enqueue or dequeue messages for network-impacting queues or directly influences message sending, it can be abstracted away.

An example of an action which is network-impacting is given in the following listing:

```plaintext
action (a_issueRequest, "a", desc="Issue a request") {
    enqueue (requestNetwork_out, RequestMsg, issue_latency) {
        out_msg.addr := address;
        out_msg.Type := CoherenceRequestType:GETX;
        out_msg.Requestor := machineID;
        out_msg.Destination.add(map_Address_to_Directory(address));
        out_msg.MessageSize := MessageSizeType:Control;
    }
}
```

Listing 6.1: Network-impacting actions

The `a_issueRequest` action queues a GETX request on the `requestNetwork_out` queue. This clearly makes it relevant for the network behaviour. This action is therefore essential to our research.

An example of actions that can be removed is given in the following listing:

```plaintext
action (x_copyDataFromCacheToTBE, "x", desc="Copy data from cache to TBE") {
    assert (is_valid(cache_entry));
    assert (is_valid(tbe));
    tbe.DataBlk := cache_entry.DataBlk;
}

action (z_stall, "z", desc="stall") {
    // do nothing
}
```

Listing 6.2: Actions without network impact
The action `x_copyDataCacheFromCacheToTBE` sets a cache entry datablock in the TBE. The action `z_stall` stalls the processing of a queue message. Both actions are not network-impacting, therefore they are non-essential to our research and can be removed.

**IMPACT ON THE MINIMIZED STATE MACHINE**

The removal of actions executed during transitions can result in empty action lists; these represent internal \( \tau \)-transitions. Internal \( \tau \)-transitions can be removed during the reduction phase.

Also, transitions whose action lists differ on only non-essential actions will become equal. Equality for transitions is defined as having equal action lists. When the action lists become equal because of this pattern, these transitions can potentially be combined by the reduction algorithms in Chapter 7.

**DEADLOCK BEHAVIOUR**

This pattern is sound, as we only remove non-essential actions. Any deadlock that occurs when non-essential actions are removed, also occur in the real system, and vice-versa.

### 6.2.2. MERGE SIMILAR ACTIONS PATTERN

The previous pattern removes all non-essential actions from the Cache Coherence state machines, and during the reduction phase any equal transitions are combined, where possible. So different transitions that use the same network-impacting actions can already be combined. However, when similar, but not exactly equal network-impacting actions are used, this optimization should also be possible. See for instance the next actions from the MI_example directory controller:

```
action (a_sendWriteBackAck, "a", desc="Send writeback ack to requestor") {
    peek(requestQueue_in, RequestMsg) {
        enqueue(forwardNetwork_out, RequestMsg, directory_latency) {
            out_msg.addr := address;
            out_msg.Type := CoherenceRequestType:WB_ACK;
            out_msg.Requestor := in_msg.Requestor;
            out_msg.Destination.add(in_msg.Requestor);
            out_msg.MessageSize := MessageSizeType:Writeback_Control;
        }
    }
}

action (l_sendWriteBackAck, "l a", desc="Send writeback ack to requestor") {
    peek(memQueue_in, MemoryMsg) {
        enqueue(forwardNetwork_out, RequestMsg, 1) {
            out_msg.addr := address;
            out_msg.Type := CoherenceRequestType:WB_ACK;
            out_msg.Requestor := in_msg.OriginalRequestorMachId;
            out_msg.Destination.add(in_msg.OriginalRequestorMachId);
            out_msg.MessageSize := MessageSizeType:Writeback_Control;
        }
    }
}
```

Listing 6.3: Similar actions in Directory controller

It is easy to see that these actions, while different, are very similar when only looking at their networking aspects. They both respond with a \texttt{WB\_ACK} acknowledgement message.
on the forwardNetwork_out queue. The differences are mainly which location the data comes from, either from an incoming message or from the memory. In the message content we are not interested, from a networking impact point of view any two transitions with only these actions would be eligible to be combined.

This can be achieved by simply renaming all similar actions to the same action name. This will make that any transitions that can be combined, will be combined during the reduction phase. Obviously this also requires that all other essential actions executed during the transitions are equal.

**IMPACT ON THE MINIMIZED STATE MACHINE**

By making actions equal, we can also make action lists equal. This means that the transitions can also become equal, as equality for transitions is defined as having equal action lists. Any equal set of transitions can potentially be combined, again depending on the state machine configuration.

**DEADLOCK BEHAVIOUR**

The similar actions are differing in the original, fully functional cache coherence protocol. As we are removing irrelevant aspects with the abstraction, what we retain are not similar, but completely equal actions. If you look at example in Listing 6.3, the difference is whether the message content is read from the memQueue_in, or from the requestQueue_in. From a network-impacting perspective we are only interested in the type of message returned, not in the message content. This also implies that the deadlock behaviour of the resulting actions are the same, so they can be combined.

**6.2.3. REMOVAL OF NON-ESSENTIAL DEVICE PATTERN**

When the DMA coherence node is deemed out of scope for model checking, we cannot simply remove all actions related with the DMA protocol in the state machines of the other devices. This would make all transitions for the DMA events internal \( \tau \)-transitions; and this is not correct, because when we remove the whole device, these transitions will never be executed anymore. Transitions not taken are different from transitions that become internal. Internal transitions can *always* be taken, while these transitions will *never* be taken.

Figure 6.5 shows a MI Directory state machine partially abstracted and reduced with the first pattern, ‘Removal of non-essential actions pattern’. When we would remove the inv_sendCacheInvalidate action from the transition between the state M and the state M_DRD,M_DRW, \(^2\) we would remove the transition for the DMA_READ and DMA_WRITE events. However, we would then combine these states, and immediately allow triggering of the PUTX event from the combined M,M_DRD,M_DWR state. In this case this looks harmless, as the M state also allows a transition on the PUTX event, other state machine configurations however would have new transitions added to the M state. These new transitions violate the sound-ness property of our patterns.

The correct way is to remove all impacted transitions directly from the coherence state machines. In Figure 6.5 we would remove all transitions for the the events DMA_READ and DMA_WRITE.

\(^2\)This is how the reduction algorithm outputs merged states.
Figure 6.5: Partially abstracted state machine with DMA transitions
The relevant transitions can be identified by inspecting the receiving queue handling functions. For instance, the following code handles the incoming DMA queue for the Directory node:

```c
in_port ( dmaRequestQueue_in , DMARequestMsg, dmaRequestToDir ) {
    if ( dmaRequestQueue_in . isReady ( clockEdge ( ) ) ) {
        peek ( dmaRequestQueue_in , DMARequestMsg ) {
            TBE tbe := TBEs [ in_msg . LineAddress ];
            if ( in_msg . Type == DMARequestType :READ ) {
                trigger ( Event :DMA_READ, in_msg . LineAddress , tbe );
            } else if ( in_msg . Type == DMARequestType :WRITE ) {
                trigger ( Event :DMA_WRITE, in_msg . LineAddress , tbe );
            } else {
                error ( "Invalid message" );
            }
        }
    }
}
```

Listing 6.4: DMA queue processing in Directory node

This code defines that on a DMARequestType :READ request a transition with Event :DMA_READ is to be handled. For the DMARequestType :WRITE type message a transition with an Event :DMA_WRITE is to be handled.

When we remove all transitions with these events, we effectively remove all behaviour linked to the DMA messages, without introducing additional internal \( \tau \)-transitions. With this pattern we can remove transitions that use network-impacting actions, something also not possible if we would use the 'Removal of not-essential actions' pattern.

We need to make sure that the events are only issued for messages coming from the device to remove. If the event definitions are shared between various devices we cannot remove the transitions, as these are also re-used by the other device.

To retain the Strongly Connected Component (SCC) feature of the cache coherence state machines, we also need to remove all states and transitions that have become unreachable by removing the transition. For the example in Figure 6.5 we would remove the M_DRD and M_DWR states, and the PUTX transitions starting from these states. Effectively we are removing the traces that were only implemented to support the removed component.

These are the states that have no incoming transitions apart from the ones we just removed, and all transitions that originate from these removed states.

The states can be identified automatically by either searching for states without incoming transitions, or by using an algorithm as used in Section 7.2.4. The latter algorithm also covers the situation that multiple states to remove form a loop, analogous to the discussion in Section 7.2.4.

Additionally, we need to remove all states that become end-states because of the removed transitions. This can be done in a similar way, for instance by using Tarjan's strongly connected components algorithm, and selecting the SCC containing a stable state, in case transition states form a loop.

\(^3\text{In this case we can use any non-transient state as initial state. The Invalid state would be a good starting state.}\)
6.3. **Have we found all abstraction patterns?**

With the three abstractions we described in this paper we cover many cases where we remove non-essential features from the cache coherence state machines. The question now arises whether we can remove all non-essential features with these three patterns.

The answer is depending on state machine we investigate. What we want to do with these patterns is to obtain only the essential complexity from the state machine; the complexity needed to implement the features that are relevant to us, the network-impacting features. What we want to remove are the incidental or accidental complexity, introduced by the state machine designers to implement features.

As an example, when we look at the MESI L1 cache state machine the concept of similar states arises from the design choice to implement load/stores, optional/mandatory and data versus Instruction fetches as separate, parallel transitions. In the simpler MI state machine we see no instance of this pattern. An alternative implementation could have put this complexity in the action implementation itself, which would have led to a simpler state machine, but more complex and less granular actions.

The design decision to handle the complexity this way gives lead to the emergency of this abstraction pattern, which effectively reverses this decision and removes (abstracts) this complexity again.

What we want to do with these patterns is to remove the complexity introduced by the design decisions the state machine designers made. This means that the amount of abstraction patterns is maybe only bound by the limits to the ingenuity the designers had when implementing these protocols.

As more complex state machines can lead to more ingenuity in the found solutions, our expectation is indeed that we might find more patterns when we investigate more complex and different coherence protocols and protocol implementations. As an example, in the discussion on the reduction of the Layer 2 cache controller in Section 8.2 we give evidence of yet another pattern. We did not document this pattern because of time constraints, otherwise we would have described four abstraction patterns in this paper.
While the abstraction process can be seen as removing unnecessary features, the reduction process is responsible for the minimization of the state machine to the most simple form, while preserving all relevant features. So what the reduction step tries to do is to morph the state machine into a similar but simpler state machine, with the same characteristics. This new state machine is supposed to be less complex, but can still be used in place of the original, abstracted state machine.

In this chapter we first determine the kind of equivalence we are interested in, after which we describe the method we use to implement the state machine reduction, while preserving this kind of equivalence.

As an input we have cache coherence state machines that, while abstracted, are still finite, single Strongly Connected Component (SCC) state machines without initial- and end states. Every state can still be reached from every other state, either directly or indirectly. For all transitions both event labels and action lists are defined, but transition equality is only determined by the action lists. As these event labels normally identify the incoming message types, this means that we will handle incoming message types handled with the same network-impacting actions as equal.

Empty action lists are seen as internal $\tau$-transitions from the originating to the destination state, so a transition that is not externally visible. While these internal $\tau$-transitions are normally not present in the provided, original cache coherence state machines, they can occur in the state machines after abstraction. They are created when all actions are removed from a transition.

Because of these $\tau$-transitions and the fact that we can get equal transitions emanating from a state for different message types, we must treat the state machines as Non-deterministic Finite Automata (NFA).

We can also state that we are only interested in the externally visible behaviour of cache coherence state machines. The model checking that these state machines are going to be used for, have no requirements on the internal structure of the state machine.

This chapter describes a number of existing techniques to reduce abstracted state machines. We also describe two contributions that were not found in existing literature. The
first is the proof that trace equivalence and (bi-)simulation equality coincide for DFAs (Deterministic Finite Automata), as we describe in Section 7.1.4. For this we did find theorems in [B+08, p.511, p.514] that this is the case for AP-deterministic systems, but without proof and not for the LTSs (Labelled Transition Systems) we use.

The second new technique we provide is the removal of superfluous initial states created by the standard NFA to DFA conversion algorithm, as explained in Section 7.2.4.

7.1. **STATE MACHINE EQUIVALENCE**

What we first need to determine is how we are going to define when two state machines are equal. We need this to be able to define what ‘preserving relevant features’ exactly means, so when the abstracted sparse state machines and the final reduced state machines are still considered equal for our purposes.

So to determine what we can safely remove from a state machine while still preserving this kind of equality, we first need to determine the kind of equality we are interested in.

What we are interested in is *behavioural equivalence*. We want to treat the cache coherence state machines as black box implementations, and want to have equality based on the traces that the state machine accepts. When the original state machine allows or generates some trace, we want the reduced state machine to allow or generate that exact same traces.

We will first look at trace equivalence and structural equivalence, after which we will look at bisimilarity type equality. For much more detail on these kinds of equivalence, see [San12].

7.1.1. **TRACE EQUIVALENCE**

Trace equivalence checks if two state machines can handle the same traces, so the same sequence of state transitions. In our context, a trace is defined as a sequence of event/action lists that are handled in order.

For instance in Figure 7.1 it can be seen that both state machines accept the string ‘ab’. This makes them trace equivalent. However, trace equivalence does not compare for which sequence of state transition traces are *not* accepted. In the left state machine however, when by accepting ‘a’ the transition to state $P_4$ is taken, the subsequent ‘b’ leads to a deadlock. These state machines are *not* behaviourally equivalent when you look at them from a system execution perspective, as only one can experience this deadlock.

However, in our case this does not present problems, as our state machines are defined as having no end states. Each alternative choice in the state machine just leads to different behaviour, and never to a deadlock.

On first look, one would say that trace equivalence is a very good match for our state machine behaviour (as even the name suggests it). However, we run into two problems with this kind of equality definition:

- Trace equivalence is difficult to determine. You would have to compare all traces possible in the original state machine for acceptance in the reduced state machine, and vice versa. This is not made easier by the fact that in principle our traces have infinite length;
• No state machine reduction algorithms are readily available that reduce according to trace equivalence.

In the next sections we show that there still are efficient methods to reduce according to trace equivalence, but these rely on specific attributes of the state machine itself.

### 7.1.2. Structural equivalence

Another type of equivalence on state machines is structural equivalence or isomorphism. Two state machines are structurally equivalent when for all states and state transitions a bijection, a reciprocal 1-1 relation, can be established.

This is a very strong kind of equivalence. Not only does this include the behaviour of a state machine into the equivalence, but also the structure of the state machines.

What this can lead to can be seen in Figure 7.2. These two state machines are trace equivalent, but not structurally equivalent. Using an equality relation like this would mean that a reduction from the Q state machine to the P state machine would not be allowed, while the behaviour is exactly equal. With these restrictions not many reductions would be possible. So we need an equality relation less strong than structural equivalence.

### 7.1.3. (Bi-)Simulation equivalence

The equivalence relation we could be looking for is something like bisimulation equivalence. Bisimulation equivalence is also called bisimilarity, and if we use this equivalence relation we are looking for reduced, simpler state machines that are bisimilar with the original state machines.

According to [San12], ‘Bisimilarity is accepted as the finest extensional behavioural equivalence one would like to impose on processes’. This means that all behaviour of the state machines is taken into account when comparing for bisimilarity.

Bisimilarity in this respect is stronger than trace equivalence, but weaker than structural
7.1. State Machine Equivalence

The state machines in Figure 7.3 are not bisimilar. This can easily be seen as no state can be found in the left state machine that has similar transitions as $Q_{2,3}$. However, we would still like to have this kind of optimization, as it removes equivalence. Additionally it provides equivalence on the behaviour of the state machine, which is the kind of equivalence that we want. So while in Figure 7.2 the state machines are structurally not equivalent, they are bisimilar. Bisimilarity is noted with the ‘tilde’ $\sim$ sign, as can be seen between the two state machines in the Figure 7.2.

Informally, two state machines are bisimulation equivalent when for all transitions from all states a similar state can be found with the same transitions, and in which the destination states of the transitions in both state machines also have the same property. In the other direction the same relation is to be found.

Formally this can be defined as:

A relation $R$ on processes is a bisimulation if whenever $P R Q$:

$$\forall \mu, P' \text{ such that } P \overset{\mu}{\rightarrow} P', \text{ then } \exists Q' \text{ such that } Q \overset{\mu}{\rightarrow} Q' \text{ and } P' R Q';$$

$$\forall \mu, Q' \text{ such that } Q \overset{\mu}{\rightarrow} Q', \text{ then } \exists P' \text{ such that } P \overset{\mu}{\rightarrow} P' \text{ and } P' R Q';$$

(7.1)

The bisimulation relation $R$ is a set of pairs of states, which have the same transitions, and for which the destinations of the transitions are also in the relation.

In Figure 7.2 for the state $P_1$ there are two states that fulfill this relation, namely $Q_1$ and $Q_3$, and vice versa the two Q states map also on $P_1$. For $P_2$ the state that is bisimulation equal is $Q_2$, which also applies in the other direction. So the bisimulation relation for this figure is:

$$R = \{(P_1, Q_1), (P_1, Q_3), (P_2, Q_2)\}$$

One of the nice things about bisimilarity is that it can be checked very efficiently, much more efficiently than trace equivalence. Where you need to check all traces in a state machine to determine trace equivalence, you only need to setup the bisimulation relation to determine bisimulation equivalence. Bisimilarity implies trace equivalence, all bisimilar state machines are also trace equivalent. Another good thing is that efficient reduction algorithms exist, also based on the above equations.

Bisimulation Equivalence or Bisimilarity

According to the Equation 7.1 the state machines in Figure 7.3 are not bisimilar. This can easily be seen as no state can be found in the left state machine that has similar transitions as $Q_{2,3}$. However, we would still like to have this kind of optimization, as it removes...
one state and one transition from the state machine. So for our purposes standard (strong) bisimilarity equivalence is still too strong. Also, standard bisimilarity does not work with internal $\tau$-transactions which we have in our abstracted state machines.

One option to solve this would be to use a weaker bisimilarity relation. Many exist that provide weaker equivalence relations. A good example of such a weaker bisimilarity relation would be branching bisimilarity, which is ‘an equivalence relation on processes that preserves the branching structure of processes’ [Bas96]. Branching bisimilarity also allows $\tau$-actions. It could very well be that a branching bisimulation variant could be applicable to our solution. We did not pursue this direction as in our case there are simpler methods to reduce our state machines. These methods are detailed in the next sections.

**Simulation equivalence**

Another type of equality we could use is a ‘similarity’ relation. This relation is noted as $\leq$ or $\geq$ and is the bisimulation relation, but only in one direction. This is a weaker form of equivalence than bisimilarity. When looking again at Figure 7.1 the Q state machine is similar to P ($Q \leq P$), but not vice versa. We can create a relation $R$ from all Q states to the P states, but have no Q equivalent for $P_4$.

When we combine this simulation relation in two directions, $P \leq Q$ and $Q \leq P$, we get simulation equivalence $P \leq \geq Q$. As an example, P and Q in Figure 7.1 are not simulation equivalent, as we cannot create a relation in both directions.

Simulation equivalence is weaker than bisimulation equivalence, because of this separate relation set for each direction. However, just like trace equivalence, standard simulation equivalence does not respect deadlock, as stated in [San12, p. 168]. Therefore, we cannot use it. ¹

### 7.1.4. Equality between trace equivalence and (bi-)similarity

What we would like to do is to compare using trace equivalence, but reduce the state machine using strong bisimulation reduction algorithms. We would then have the needed form of equality, while also being able to leverage the simple bisimulation based algorithms to reduce a state machine. For AP-Deterministic state machines this is possible, as [B+08, p.579] state, “when we have an AP-deterministic state machine, bisimulation, simulation and trace equivalence coincide”.

This means that if we can provide the same proof for deterministic Labeled Transition System (LTS) state machines, we can use (strong) bisimulation equivalence and reduction algorithms to reduce the resulting state machine. The resulting minimized state machine

¹As with bisimulation equivalence, there are also other types of simulation equivalence that could be more appropriate for us. We do not pursue these alternatives as we assume these are more complex than the solution below
will not necessarily be (bi-)simulation equivalent to the original state machine, only to the already abstracted variant that we converted into a Deterministic Finite Automaton (DFA) with the NFA to DFA step.

Bisimulation reduction algorithms reduce the size of state machines while retaining bisimulation equivalence. If we can make use of the above, the algorithms will also retain trace equivalence, as these are equal.

The question is now, can we make the above statement also for the type of action-oriented LTSs that the cache coherence state machines are? This is indeed the case as proven below, because for DFAs we can also determine that these equivalence relations are equal.

DFAs are deterministic action-oriented LTSs which have only single transitions for each action starting from a state. For any action you never have a choice in destination states, as for each action/label a DFA at most only this single transition can be chosen. Also, DFAs do not have internal \( \tau \)-transitions.

This is in contrast to Non-deterministic finite automatons (NFAs), which can have both choices and internal \( \tau \)-transitions. The abstracted cache coherence state machines we want to reduce are NFAs.

**Definition**  
We define a DFA as:

\[
DFA = \langle Q, L, T \rangle
\]

For our purposes a DFA is defined as a combination of a set of states \( Q \), a set of labels \( L \) and a transition relation \( T \).

For these DFAs we need to define what we exactly mean with traces and paths through a DFA.

**Definition**  
Let \( p \) be a (state of a) labeled transition system, with:

1. \( \text{traces}(p) =_{\text{def}} \{ \sigma \in L^* \mid p \xrightarrow{\sigma} \} \)

2. \( \text{paths}(p) =_{\text{def}} \{ (\sigma, \pi) \mid \sigma \in L^*, \pi \in Q^*, \sigma = [\mu_1, \mu_2, \ldots, \mu_m], \pi = [p, p_1, \ldots, p_m], p \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_m} p_m \} \)

A single trace is defined as a Kleene closure over the set of labels. This results in a list of labels, length 0 or more, which is satisfiable by the DFA starting from state \( p \).

A single path is defined as a list of transitions, combined with the list of the states the path visits. This results in a pair of lists, which are satisfiable by the DFA starting from state \( p \).

If we have two DFAs, what we want to prove is that iff they are trace equivalent, that they are also simulation equivalent and bisimulation equivalent:

**Lemma 1**

\[
\text{traces}(DFA_1) = \text{traces}(DFA_2) \quad \iff \quad DFA_1 \leq DFA_2 \quad \iff \quad DFA_1 \sim DFA_2
\]

**Proof 1**  
We ignore the concept of initial and terminal states here, as these are not present in the state machines we use, but they are also not relevant for this proof. The starting state of a trace can be any state in the LTS.

---

2 Adapted from [Tre08, p.9], L labels are equivalent to action lists.
For DFAs we can see that for each trace \( \epsilon \in \text{traces}(p) \) there is a unique path \( \epsilon \in \text{paths}(p) \). This is because of the lack of choices and internal \( \tau \)-states. If we compare two DFAs, named \( DFA_1 \) and \( DFA_2 \), and they are trace equivalent, by definition there is for each trace in \( DFA_1 \) a corresponding, equal trace in \( DFA_2 \). This means that these DFAs also have equal paths, where only the state(names) themselves differ. If we create a relation \( \mathcal{N} \) which maps the states between the DFAs, we can state the following:

\[
\text{traces}(DFA_1) = \text{traces}(DFA_2) \iff \exists \mathcal{N} \text{ paths}(\mathcal{N}(DFA_1)) = \text{paths}(DFA_2)
\]

For the DFAs in Figure 7.4 the relation \( \mathcal{N} \) would be:

\[
\mathcal{N} = \{(P_a, Q_a), (P_b, Q_b), (P_c, Q_a), (P_d, Q_b)\}
\]

As we make this definition for all traces in the DFA, and additionally for all paths, the path equivalence covers all outgoing transitions for all states. For all states in \( P \), there is a state in \( Q \) with the exact same transitions, to destination states with the same equality. Otherwise not all paths could be equal.

This path equivalence is exactly the equality relation that we cannot make for NFAs, as the choices and internal \( \tau \)-transitions can cause multiple paths for a trace.

An example can be seen in left part of Figure 7.3, where we have two paths with the same \([\mu_1]\) trace. The right state machine is the DFA variant, which does not have these different paths. Also, not shown in this example, internal \( \tau \)-transitions could also have been part of traces in the NFA on the left, which would not have been visible in the trace, but could have resulted in extra states and transitions in the paths. These extra states and transitions, and the choices one can make, make that any trace in an NFA can result in multiple paths.

Simulation equivalence \( \leq \geq \) for two LTSs is defined as follows:

\[
\forall \mu, P' \text{ such that } P \xrightarrow{\mu} P', \text{ then } \exists Q' \text{ such that } Q \xrightarrow{\mu} Q' \text{ and } P' \not\sim Q';
\]

\[
\forall \mu, Q' \text{ such that } Q \xrightarrow{\mu} Q', \text{ then } \exists P' \text{ such that } P \xrightarrow{\mu} P' \text{ and } P' \not\sim Q';
\]

What this definition describes is when \( P \leq \geq Q \), then for all states in \( P \), we can find a state in \( Q \) with the exact same transitions. If we have two trace equivalent DFAs, we know that that is the case, as it is shown above that these have equal paths, and so the the state relation relation \( \mathcal{N} \) can be used as \( \mathcal{R} \) and \( \mathcal{S} \) in the similarity relation. So this means:

\[
DFA_1 \leq \geq DFA_2 \iff \text{traces}(DFA_1) = \text{traces}(DFA_2)
\]

As \( \mathcal{R} \) equals \( \mathcal{S} \) (as both equal \( \mathcal{N} \), the trace equivalence mapping above), the similarity relation for \( \leq \geq \) above is equal to the bisimilarity relation \( \sim \), as defined in Equation 7.1. So therefore we can state that in our case similarity and bisimilarity coincide:

\[
DFA_1 \leq \geq DFA_2 \iff DFA_1 \sim DFA_2
\]

By combining 7.3 and 7.4 this result we can conclude that for deterministic action oriented state machines (so DFAs), trace-equivalence, similarity equivalence and bisimilarity equivalence coincide. This means that we have proven Lemma 1.
This result is interesting, because it means that if we convert our NFA coherence state machines to a DFA, we can then use standard strong bisimulation reduction to minimize this DFA, and this minimal DFA remains trace equivalent to the original NFA. Additionally, all bisimulation defined attributes of this state machine, as eventual deadlock behaviour, etc. are retained.

This result is only valid for DFAs, we first need to convert our NFA state machine with internal states to a trace equivalent DFA. Luckily, standard algorithms are available to assist us in that. How this is done is the subject of the next section.

7.2. NFA TO DFA CONVERSION

For the conversion from an NFA to a DFA we use the algorithm as described in detail by Linz in [Lin11, p.59], which also provides proofs of the correctness of the algorithm steps. Linz’s algorithm basically creates combination states every time a choice is made in the NFA, or an internal transition is made. This way it constructs a trace equivalent DFA without any choices and internal \( \tau \)-transitions. This is the well-known subset construction algorithm, combined with the also standard \( \tau \)-reduction algorithm.

The DFA resulting from this conversion can have more states, in theory up to \( 2^{\mid \text{NFA} \mid} \) states, where \( \mid \text{NFA} \mid \) is the set of states in the original NFA. However, this is dependent on the topology of the NFA, and we assume the topology of the state machines we work with is such that in the end, after we also reduced the state machine, we will have a smaller state machine as a result.

7.2.1. SUBSET CONSTRUCTION

The subset construction (also called powerset construction) is an algorithm which is composed of the following steps (taken from [Lin11, p.59]):

1. Create a graph \( G_D \) with vertex \( \{q_0\} \). Identify this vertex as the initial vertex.

2. Repeat the following steps until no more edges are missing:

   (a) Take any vertex \( \{q_i, q_j, \ldots, q_k\} \) of \( G_D \) that has no outgoing edge for some \( a \in \Sigma \).

   (b) Compute \( \delta^*_N(q_i, a), \delta^*_N(q_j, a), \ldots, \delta^*_N(q_k, a) \).

   \( \delta^*_N(q_i, a) \) is the transition function which computes the destination state set \( \{q_l\} \) for \( q_i \) and \( a \) as: \( \delta^*_N(q_i, a) = \{q_l | q_i \xrightarrow{a} q_l \} \).

   (c) If \( \delta^*_N(q_i, a) \cup \delta^*_N(q_j, a) \cup \cdots \cup \delta^*_N(q_k, a) = \{q_l, q_m, \ldots, q_n\} \), create a vertex for \( G_D \) labeled \( \{q_l, q_m, \ldots, q_n\} \) if it does not already exist.

   (d) Add to \( G_D \) an edge from \( \{q_i, q_j, \ldots, q_k\} \) to \( \{q_l, q_m, \ldots, q_n\} \) and label it with \( a \).

3. Every state of \( G_D \) whose label contains any \( q_f \in F_N \) is identified as a final vertex.

4. If \( M_N \) accepts \( \lambda \), the vertex \( \{q_0\} \) in \( G_D \) is also made a final vertex.

For us the only interesting step is Step 2. We have no initial states, so we cannot create a vertex \( \{q_0\} \). We also have no final states, so also the last two list steps are also not in scope for us. The issues encountered with determining the state to start with if we have no DFA initial state are covered in Section 7.2.4.
7.2.2. **TAU-A REDUCTION**

The tau-a reduction algorithm used to remove all $\tau$-transitions is combined with step 2.b above. When computing $\delta^*_N(q_i, a)$, we also add all states which are reachable with $\tau$ transitions and which have an ‘a’ transition themselves. So this results in the following extension to the transition function:

$$\delta^*_N(q_i, a) = \{ q_i | q_i \xrightarrow{\tau^* a} q_l \}$$

where $\tau^*$ is 0 or more $\tau$ transitions

As already mentioned, in the cache coherence state machines the $\tau$-transitions are transitions with empty action lists.

The result of the this delta calculation is a set of 0 or more NFA destination states for transitions with action $a$, which together with the result of the calculation of the other NFA states in the vertex $\{ q_i, q_j, \ldots, q_k \}$ result in the new destination state vertex $\{ q_l, q_m, \ldots, q_n \}$.

As a side note, the current version of the tool used to validate the simplification process completely removes the intermediate states and transitions from the state machine. This is the default implementation of the above algorithm, but must be taken into account when validating the result.

7.2.3. **HASSELL IMPLEMENTATION**

The haskell implementation of this algorithm is as follows. We keep a to-do list of states still to process, and seed it with the startstate of the first transition. This startstate we take as the ‘random’ initial state, as defined in Section 7.2.4.

As long as the to-do list of states is not empty, we repeat the following actions:

1. We take the state $s$ from the top of the list
2. We do the tau-a reduction for $s$. To be able to automate the algorithm, we slightly modify the tau-a reduction algorithm. We calculate all states reachable from $s$ using $\tau$-transitions, using:

$$\text{startStates} (s) = \{ p | s \xrightarrow{\tau^*} p \}$$

3. Determine all actions originating from these states, using:

$$\text{actions} (\text{startStates}) = \{ a | p \xrightarrow{a} \ldots \land p \in \text{startStates} \}$$

4. determine for each action $a \in \text{actions}$ the transition by combining the NFA deststates:

$$\text{destStates} (a, \text{startStates}) = \{ q | p \xrightarrow{a} q \land p \in \text{startStates} \}$$

5. create for each action a new edge or DfaTransition that links $s$ to the new destState, created by combining the destStates just calculated for that action. Add this new DfaTransition value to the DFA to create.

6. if the new destState is not already seen (in the to-do list or already present in the created DFA), add the destState to the to-do list for later processing

When this process ends with an empty to-do list, we have gone through all states we can reach from the original initial state, and have created a DFA, trace-equivalent with the original NFA, as proven by Linz in [Lin11, p.59].
7.2. IMPACT SELECTION OF INITIAL STATE

What we needed to add to use the algorithm described in [Lin11, p.59] was an initial state. As we have not defined an initial state, we will start with a random state, but this will have impact on the eventual DFA state machine. This can be seen in Figure 7.3. The right state machine is in fact the DFA version of the left state machine. You can see that the states $P_2$ and $P_3$ are subsumed in a new state here labelled $Q_{2,3}$. Now suppose these are parts of larger state machines. It could be that if you start the subset reduction algorithm with $P_2$ you would add $P_2$ also to the DFA state machine as an extra state.

These extra states are easy to identify. In the SCC cache coherence state machines we analyse we can always reach each state from each other state, directly or indirectly. However, if we use $P_2$ from Figure 7.3 as the initial state, it could be that this is not a destination of a transition from any state in the rest of the DFA, because these transitions would lead to the combined state labelled $Q_{2,3}$. This is also the reason that a Cache Coherence Invalid state is not chosen as the default initial state. It would suffer the same problem as any other state, as this state too could be combined during the subset construction. Choosing a ‘random’ state will result in the same outcome, and makes the algorithm input simpler.

To correct this startup behaviour, we have developed a correction algorithm to find these extra initial states, to remove them again from the created DFA.

The question is now how can we determine these extra states and transitions. As an example, see Figure 7.5. In this figure we have calculated the resulting DFA (on the right) when picking state $P_1$ as the initial state. As a counter example, had we started with $P_3$ as the initial state, we would not have the states $P_1$ and $P_2$ in the resulting DFA. This means that $P_1$ and $P_2$ are extra intermediate states, which we must remove from the result. As they both have incoming transitions, this shows that a naïve algorithm that would only check for states without incoming transitions will fail.

For this state machine one would be tempted to think that a strategy to be ‘smart’ when

---

2If it did get an incoming transition from the rest of the DFA, it would be because of a single non-combined transition from another state in the DFA itself. Then it would obviously rightfully be part of the final DFA, but this would have shown in Figure 7.3 as an extra incoming transition.
selecting a good starting state from the NFA state would work. A counter example that also a smart selection would not work is shown in Figure 7.6. There we see a variant on the previous figure which is slightly modified with an extra state. This makes that the ideal resulting DFA does not re-use any original NFA states anymore, and therefore the strategy to do a ‘smart’ selection of the initial state is bound to fail.

However, we know that the final DFA is a SCC, therefore all states which have to become part of the resulting DFA are all directly or indirectly reachable from each other state. This is because the original NFA is a SCC, and so should the DFA be to retain trace equivalence. This is why we have the one-way transitions from the ‘intermediate’ part to the SCC part of the DFA. See for an example of this Figure 7.5, between $P_2$ and $P_3$. We know that the last state we construct is by definition part of the SCC. Therefore, when we determine the states reachable from this last state, we will get only the states in the SCC. This way we will reliably be able to separate the SCC from the intermediate states. This works regardless with which state we started the original NFA to DFA algorithm with, as we will show in the rest of this section.

Let $DFA_{rough} = (Q_{rough}, L, \delta_{rough})$ denote the original DFA with the extra initial states. We can then find $DFA_{final} = (Q_{final}, L, \delta_{final})$, which is the SCC part of the calculated $DFA_{rough}$ with the intermediate starting states and their transitions removed.

State $q_{rough, last}$ is the last state added to $DFA_{rough}$, and we can use it to construct $Q_{final}$:

$$Q_{final} = \{ q' | \exists \sigma \in L^*: q_{rough, last} \xrightarrow{\sigma} q' \}$$

The transition relation $\delta_{final}$ contains all transitions from $\delta_{rough}$ which originate from a state in $Q_{final}$. Combining both $\delta_{final}$ and $Q_{final}$, we have $DFA_{final}$.

What we want to retain over all NFA to DFA conversion actions is that the accepted traces must not change. We formalize this with the Lemmas below.

We want to retain trace equivalence when we convert the NFA to a DFA:

**Lemma 2**

$$traces(NFA_{orig}) = traces(DFA_{rough})$$
7.2. NFA TO DFA CONVERSION

**Proof 2**  The equivalence between $NFA_{\text{orig}}$ and $DFA_{\text{rough}}$ is defined by the NFA to DFA conversion algorithm. The proof for this algorithm can be found in [Lin11, p.59].

Similarly, we want to retain the trace equivalence when removing the additional states in $DFA_{\text{rough}}$.

**Lemma 3**

$$traces(DFA_{\text{rough}}) = traces(DFA_{\text{final}})$$

**Proof 3**  $DFA_{\text{final}}$ is the SCC part of $DFA_{\text{rough}}$. Any difference between $DFA_{\text{rough}}$ and $DFA_{\text{final}}$ is caused by the choice of the initial state, where each of these states (if not in $DFA_{\text{final}}$) is also present in the SCC, but then combined with other NFA states. All traces originating from the removed initial states (if any) can also be found in the resulting $DFA_{\text{final}}$, which is the SCC part of $DFA_{\text{rough}}$. The SCC transitions will emanate from the combined NFA states.

Because of this, we can state that there are no extra traces in $DFA_{\text{rough}}$ that are not in $DFA_{\text{final}}$. In the other direction, as $DFA_{\text{final}}$ is the SCC subset of $DFA_{\text{rough}}$, there are no additional traces in $DFA_{\text{final}}$ that are not in $DFA_{\text{rough}}$. Therefore we can state that they are equal, and $DFA_{\text{rough}} = DFA_{\text{final}}$.

Combining Lemma 2 and Lemma 3 results in the following:

**Corollary 1**

$$traces(NFA_{\text{orig}}) = traces(DFA_{\text{rough}}) = traces(DFA_{\text{final}})$$

**ALGORITHM TERMINATION**

With this algorithm in place we are left with one question: does this initial state algorithm terminate in all cases when a random initial state is chosen? So given an finite state machine (NFA) in which all states can be reached from all other states? So in other words, will the eventual DFA be constructed. This answer is positive, as the below proof shows:

**Lemma 4**  The removal of the extra initial startup states algorithm always terminates.

**Proof 4**  Suppose we have an NFA $N$, resulting in a DFA $D$ after tau-a reduction and subset construction, with optionally extra startup states. When the original NFA to DFA algorithm reaches a state that is destined for $D$, it will subsequently only build states for $D$. We start with the last state, which is therefore always part of $D$. As $D$ is to be a single SCC, we can reach any state from the any other state, and so any state not reachable from the last state is not part of $D$. The algorithm is such that we will never re-visit an already found or generated state, so after evaluating $|D|$ states we will have constructed $D$ fully. So the algorithm will eventually always terminate with the creation of the DFA.

The initial state removal only visits states in $D$, and so is not impacted by any amount of initial startup states that are generated outside of $D$.

The conclusion is that even when we do not start the tau-a reduction and subset construction from a state in $D$, we will still create the final DFA after executing the algorithm a finite amount of time, proportional to $|D|$.

In normal cases we would expect to have only one or two extra startup states before a DFA state is constructed.
7.3. STRONG BISIMULATION MINIMIZATION

After we have done the NFA to DFA transition, we have the DFA needed to perform the bisimulation reduction step. This reduction basically works by partitioning the set of DFA states in equal partition blocks, where each state in a partition has transitions to other states also in the same destination partition. This partitioning must first start by creating two or more partitions. This can instance be done, based on whether states are final or not [Lin11, p.63] or by the labels for Kripke structures [B+08, p.476]. After this initial partitioning into partition blocks, these blocks are then further refined.

After executing the partitioning algorithm, when the DFA is completely refined, we have an end result with blocks of states where all states have the same labeled transitions. All transitions with the same label are to states in the same destination block. This closely resembles the bisimilarity relation from Equation 7.1, where the transitions for states also must be to ‘equal’ states. So in effect what we have created is a form of the bisimulation relation $\mathcal{R}$.

This completely refined set of blocks can in turn be used to generate the minimized DFA we would like to achieve, by replacing each block with a single state. The transitions from the block are replaced by transitions from this state, and the destination state of the transition is the state in which the destination block is transformed.

We will follow the algorithm as described in [Lin11, p.63], where we - as we have no final states - assume a demonic completion of all non-existing transitions for actions to a single ‘virtual final state’. This virtual final state represents the deadlock occured when a not modeled action is encountered, this allows us to start the partitioning algorithm. This way we can use the algorithm described by Linz, and refer to the proof in that document. See [Tre08, p.17] for an explanation on demonic completion.

Linz divides the algorithm in a mark and a reduce part, conforming to the partitioning and merge steps described in the previous paragraphs. The mark and reduce algorithms are explained in the following sections.

7.3.1. MARK PHASE

As described by Linz, the mark part consists of the following activities on the DFA $M = (Q, \Sigma, \delta, q_0, F)$:

1. Remove all inaccessible states;

2. Consider all pairs of states $(p,q)$. If $p \in F$ and $q \notin F$ or vice versa, mark the pair $(p,q)$ as distinguishable;

3. Repeat the following step until no previously unmarked pairs are marked. For all pairs, $(p,q)$ and all $a \in \Sigma$, compute $\delta(p,a) = p_a$ and $\delta(q,a) = q_a$. If the pair $(p_a,q_a)$ is marked as distinguishable, then mark $(p,q)$ as distinguishable.

Step 1 is not used as we will have as output from the NFA to DFA transformation a SCC where all states are reachable from all other states. This obviously implies we have no inaccessible states. Step 2 we will do, but we will use our demonic completion to create the ‘virtual final state’ as described above.
7.3. Strong bisimulation minimization

We will use the transition for an action to this virtual final state, so whether the action exists for the state, as the differentiator. How this is implemented is described below.

**SPLIT ON ACTIONS**

For our code we will separate Step 3 into two sub-steps. We will first distinguish all states depending whether they have a transition labeled with a certain action list. We use the fact that not all states have transitions for all action lists, so we can use that to start. So we take the first action list, and partition all states according to whether they have a transition defined for that action list. This we repeat for all other present action lists. This should result in partitions of states with equal action lists.

So we partition DFA $M$ according to the following equation, where action(s) returns the actions which have transitions emanating from state $s$:

$$\text{partitionOnActions}(M) = \{ \{ s_j \mid \forall s_j \in M \text{ actions}(s) = \text{actions}(s_j) \} | s_i \in M \}$$

This formula partitions DFA $M$ into a set of partition blocks which are themselves sets of states with all equal action lists.

**SPLIT ON DESTINATION PARTITION**

The second sub-step divides the partitions we get again, based on the destinations of these actions. We know that all actions in the partitions have transitions for the same actions, as we just partitioned them on this in the previous step. We split each partition block further, based on the partition block each transition destination for an action is part of. This implementation works as follows:

When partition block $p$ contains more than one state, we take the first state from the partition $p$, $s_1$ and for each action emanating from this state we search the destination state $s_{1,a}$, so that $s_1 \xrightarrow{a} s_{1,a}$ with $a \in \text{actions}(s_1)$. We do the same for the other states in $p$, $s_i$, where $2 \leq i \leq |p|$. Their destinations are calculated as $s_i \xrightarrow{a} s_{i,a}$.

We then can do the following split, based on the partition of the destination states:

$$p_{in} = \{ s_i \mid \forall a \in \text{actions}(s_1) \text{ partition}(s_{i,a}) = \text{partition}(s_{1,a}) \}$$

$$p_{out} = \{ s_i \mid \exists a \in \text{actions}(s_1) \text{ partition}(s_{i,a}) <> \text{partition}(s_{1,a}) \}$$

The partition $p_{in}$ now contains the states with the same destination partition per action for all states, while $p_{out}$ contains the states that have destinations to other partitions for at least one action. We repeat this split action again for $p_{out}$, as the states in this partition can also still be eligible to be split further.

We repeat this second sub-step with all partitions, and repeat with all partitions until the full partition set becomes stable, i.e. no partitions are split anymore.
7.3.2. Reduce Phase
Following Linz also for the reduce phase, given DFA \( M = (Q, \sum, \delta, q_0, F) \), we will construct a reduced dfa \( \hat{M} = (\hat{Q}, \sum, \hat{\delta}, \hat{q}_0, \hat{F}) \) as follows:

1. Use the procedure mark to generate the equivalence classes, say \( \{q_i, q_j, \ldots, q_k\} \) as described above;
2. For each set \( \{q_i, q_j, \ldots, q_k\} \) of such indistinguishable states, create a state labeled \( ij \ldots k \) for \( \hat{M} \);
3. For each transition rule of \( M \) of the form \( \delta(q_r, a) = q_p \), find the partition to which \( q_r \) and \( q_p \) belong. If \( q_r \in \{q_i, q_j, \ldots, q_k\} \) and \( q_p \in \{q_l, q_m, \ldots, q_n\} \) we add to \( \hat{\delta} \) a rule \( \hat{\delta}(ij \ldots k, a) = lm \ldots n \);
4. The initial state \( \hat{q}_0 \) is that state of \( \hat{M} \) which contains the 0
5. \( \hat{F} \) is the set of all states whose label contains i such that \( q_i \in F \).

For our use we can ignore the last two steps, as we have no initial or final states. Step 1 is described in the previous paragraph, what remains is walking through the list of transitions we have, and convert their start- and endstates to simple states, one state representing each partition block. This creates a DFA with states for all partition blocks, effectively de-duplicating the state machine of bisimilar equal states.

7.4. Do We Generate the Most Minimal Solution?
We have proven that we create a sound and complete solution in the preceding sections. However, the question remains whether we generate the optimal, minimal solution.

Our hypothesis is that this could be the case, because of the combination of \( \tau \)-reduction and subset construction steps on the one hand, and the strong bisimulation reduction on the other hand.

The \( \tau \)-reduction and subset construction steps create a DFA, which can be seen as the semantically simplest way to implement the traces already present in the original NFA. However, there is a cost involved, as these algorithms can create a structurally complex solution, with potentially up to \( 2^{|N|} \) states.

On the other hand, the strong bisimulation reduction algorithm should find the structurally most simple solution while retaining the state machine semantics. The combination of these three algorithms will then result in the semantically and structurally most simple DFA.

We leave the proof (or disproof) of the conjecture that this is the minimal solution as future research, as there could be NFAs or differently constructed DFAs that are even simpler. For NFAs we have documented evidence for a potentially simpler state machine with an internal \( \tau \)-transition in Section 8.2.

\(^4\)this is the exact line from Linz, we assume the ‘0’ stands for the initial state in the DFA.
EXPERIMENTAL RESULTS

This chapter describes the results we obtained when applying the patterns and algorithms described in respectively Chapter 6 and Chapter 7. We have investigated two cache coherence protocols, the MI_example protocol and the MESI protocol, as provided in the gem5 distribution.

The results are obtained by examining the code as it existed on August 26, 2014. To replicate the results the codebase version from this date should be used.

8.1. THE MI PROTOCOL

Figure 8.2 on page 51, 8.3 and 8.4 show the original state machines of the gem5 simulator (gem5) MI cache coherence protocol. Figure 8.5 and 8.6 show the minimized state machines, after processing. The patterns were used with the MI definitions as found in Appendix A.1. Given its initial size, we did not attempt to minimize the DMA coherence device state machine.

The reduction results we obtained for the MI protocol minimization are:

Table 8.1: MI minimization results

<table>
<thead>
<tr>
<th>State Machine</th>
<th>Original States</th>
<th>Original Transitions</th>
<th>Minimized States</th>
<th>Minimized Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI Cache</td>
<td>7</td>
<td>43</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>MI Directory</td>
<td>10</td>
<td>44</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>MI DMA</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>totals</td>
<td>20</td>
<td>91</td>
<td>11</td>
<td>19</td>
</tr>
</tbody>
</table>

By applying our minimization process, we obtained the results as shown in Table 8.1. These results were obtained by applying the abstraction pattern ‘Removal of non-essential actions’ to the state machines, as described in Section 6.2.1. As can be seen in Figure 8.6 on

1This date is chosen because actions used in the supplied JSON state machine definition files were removed by commits on November 26 that year. This date is the date the last stable code version was generated which is in sync with the supplied JSON files. It is assumed that the JSON files were generated from this version. See also http://repo.gem5.org/gem5
page 54, we could reduce the MI Directory state machine with at least 2 additional transitions and one additional state when applying the ‘Remove non-essential devices’ pattern, as described in Section 6.2.3. This pattern would then be used to remove the transitions associated with the events DMA_READ and DMA_WRITE. In the MI state machines we could not find opportunities to apply the ‘Merge similar actions’ pattern.

In the directory state machine the functionality has been implemented to maintain the ownership of the cache-lines. This is done with two actions, e_ownerIsRequestor and c_clearOwner; these are used to maintain the owner of the cache-line. This ownership is subsequently used to forward GETX requests to when the directory controller is not the owner of the cache-line.

The original transitions we obtained by counting the number of transitions in the JSON state machine definition files that were provided\(^2\). The original states were counted in the state_declarations structure in the SLICC sourcecode files. We completely removed the MI DMA controller.

## 8.2. The MESI Protocol

Figure 8.1 shows the gem5 MESI component and network configuration. The original state machines for the gem5 MESI protocol are shown for the Layer 1 cache in Figure 8.7 on page 55, for the Layer 2 cache in Figure 8.8, for the Directory controller in Figure 8.9. The DMA controller had a similar state machine as the MI protocol in Figure 8.4.

As with the MI protocol, these results were obtained by applying the single abstraction pattern ‘Removal of non-essential actions’ to the state machines, as described in Section 6.2.1. We again did not attempt to minimize the DMA device for the same reason.

---

\(^2\)The state lists on the [http://www.m5sim.org](http://www.m5sim.org) website are not complete.
The reduction results we obtained for the MESI protocol minimization are:

Table 8.2: MESI minimization results

<table>
<thead>
<tr>
<th>State Machine</th>
<th>Original States</th>
<th>Original Transitions</th>
<th>Minimized States</th>
<th>Minimized Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI L1 Cache</td>
<td>15</td>
<td>146</td>
<td>8</td>
<td>43</td>
</tr>
<tr>
<td>MESI L2 Cache</td>
<td>17</td>
<td>155</td>
<td>14</td>
<td>45</td>
</tr>
<tr>
<td>MESI Directory</td>
<td>10</td>
<td>44</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>MESI DMA</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>totals</td>
<td>45</td>
<td>349</td>
<td>27</td>
<td>96</td>
</tr>
</tbody>
</table>

These results were obtained by executing the minimization process with the impacting classifications as defined in Appendix A.2. We removed all actions with a classification of ‘No’ or ‘DMA’.

The Layer 1 cache controller state machine minimization outcome can be seen in Figure 8.10 on page 58. One obvious feature of this state machine are the parallel transitions. These are caused by the fact that the L1 cache has a number of similar features from a network-impacting point of view. These are e.g. instruction versus data caches and prefetches and mandatory fetches. These have their own actions, as their non-networking impact is different. Applying the ‘similar actions’ pattern from Section 6.2.2 would combine these transitions.

Another interesting feature of the reduced L1 cache state machine is that the Exclusive state and the Modified state seen not to be related. This is counter intuitive, as one of the feature of the MESI protocol is that one can ‘silently’ move from the E to the M state. In the reduced state machines the states and transitions that are removed because of the internal \( \tau \)-transitions are really removed from the state machine. The E state in reality is the combined E and M state, where only one of the Inv transitions is unique to the E state. The others are combined transitions, and appear both for the E and M states.

This is an example where an NFA could lead to a more compact state machine. Leaving the internal \( \tau \)-transition in place would allow for all non-exclusive transitions now linked to the E state to be removed, as these transitions could be re-used from the M state. This would allow for an additional simplification of the state machine.

A future improvement to the tool could be to display the removed internal states and event on the transitions from where they were removed. This would give more insight and potential to validate the simplification process.

The minimized Layer 2 cache controller state machine is shown in Figure 8.11 on page 59. Ideally we would like to further reduce this state machine by splitting it for Layer 1 and Layer 2 network functionality. This would allow the separate investigation and modeling of the two layers. In this implementation this is not possible by removing actions, as the actions to send response messages to the L1 cache controller are also used to send responses to the Directory and DMA controller. We also cannot apply the ‘Remove non-essential devices’ pattern from Section 6.2.3 here. When this pattern is used to remove device activities, we remove full traces between stable states. This we do not want in this case, as the interaction with the L1 cache is combined with Directory controller cache interaction in single
traces.

As an example, when the L2 cache-line is in the Invalid (NP) state, the block is not present in this cache. When a GetX request is received from an L1 cache controller for this cache line, the L2 state machine sends out a fetch request (a_issueFetchToMemory) to memory and moves to the IM state. After receiving a Mem_data event from the directory controller, the data is sent back with exclusive permission to the requesting L1 cache with the action (ee_sendDataToGetXRequestor). Also, the L2 cache state machine then moves to the MT_MB, SS_MB state, as seen in the reduced state machine in Figure-8.11. Only after the L1 cache Exclusive Unblock response is received will the state machine move to the stable 'MT' state.

We obviously cannot remove any transitions from this sequence, this would break the functionality. What we want to do here is make the transitions for events from the removed device internal \( \tau \)-transitions. This is an additional pattern whose necessity is caused by the design / implementation decision to use a shared response network.

The Directory controller again provides evidence for the applicability of the Section 6.2.3 to remove the states DMA_READ || DMA_WRITE that are associated with the DMA controller.

In contrast to the MI protocol, in the MESI protocol the directory controller does not maintain the cache-line owner. This is delegated to the L2 cache controller. The L2 cache controller maintains a list of sharers, controlled with the actions marked with ‘Share’ in the L2 action Table A.5 on page 73.
Figure 8.2: Original gem5 MI cache controller state machine
Figure 8.3: Original gem5 MI directory controller state machine
Figure 8.4: Original gem5 MI DMA controller state machine

Figure 8.5: Minimized gem5 MI cache controller state machine
Figure 8.6: Minimized gem5 MI directory controller state machine
Figure 8.7: Original gem5 MESI Layer 1 Cache controller state machine
Figure 8.8: Original gem5 MESI Layer 2 cache controller state machine
Figure 8.9: Original gem5 MESI directory controller state machine
Figure 8.10: Minimized gem5 MESI Layer 1 Cache controller state machine
Figure 8.11: Minimized gem5 MESI Layer 2 cache controller state machine
Figure 8.12: Minimized gem5 MESI directory controller state machine
CONCLUSIONS AND FUTURE WORK

We conclude this thesis by discussing our findings. We list the contributions our research has made to model checking of cache coherence protocols, and methodically creating abstractions of Labelled Transition Systems. Then we will summarize the answers to the research questions from Section 3.2. We also suggest a number of areas for further research.

9.1. DISCUSSION

During our research we found that the minimization of state machines can best be done in two phases. The first abstraction phase consists of the removal of all non-relevant features from the state machines. To implement this we defined three patterns, which we used to remove non relevant features from the cache coherence state machines we investigated. This resulted in an NFA with internal $\tau$-transitions for state transitions where all features were removed, and equal transitions where all differing features were removed. All cache coherence protocol state machines are a single strongly connected component (SCC) without initial and final states. The abstraction patterns retain this Strongly Connected Component (SCC) structure and the lack of initial and final states.

As defined in the future works in Section 9.3, we did not pursue the identification of patterns further. Other state machines cache coherence state machines could reveal additional abstraction patterns. In the MESI L2 cache cache coherence protocol we identified an additional opportunity for a new abstraction pattern.

The second reduction phase consists of algorithmically shrinking the NFA state machine by applying strong bisimulation reduction. This reduction phase minimizes the state machine without changing its traces, so the end result of this step retains trace equivalence to the original abstracted NFA resulting from the abstraction. This also implies that the resulting state machine is also a single SCC without initial and terminal states.

We convert the NFA state machine to a DFA state machine with a modified subset construction algorithm. We modified the subset construction algorithm, as we could only find algorithms for NFA state machines with initial states, used to start the algorithm. The modification we developed consists of an added post processing step which removes any artifacts added by the selection of the initial state.
After this subset construction step we have a SCC DFA for which we can apply strong bisimulation reduction. Strong bisimulation reduction algorithms require that the state machines do not have internal $\tau$-transitions. We removed these during the subset construction. We have proven that for DFAs trace equivalence and bisimulation equivalence (bisimilarity) coincides. This means that two DFAs are trace equivalent iff they are bisimulation equivalent. The bisimulation reduction algorithm retains bisimulation equivalence so according to the proof it also retains trace equivalence.

We have not investigated whether this combination of algorithms is minimal. The abstracted state machine could be reduced into smaller state machines by applying alternative algorithms e.g. use weak- or branching bisimulation reduction.

### 9.2. Answers to Research Questions

In this section we provide answers for the research questions we devised at the start of our research.

1. **What patterns can be found that can be applied to reduce cache coherence state machines, creating sound abstractions?**

   We found three patterns that can be used to abstract away unneeded features of the cache coherence state machines, as described in Chapter 6. We also found evidence of additional patterns. We also found a sequence of algorithms that converts the such abstracted state machine to a reduced size. These algorithms are described in Chapter 7 and reduce the size of the state machine without changing the state machine semantics.

2. **Which state machine aspects determine the applicability of the minimization process?**

   The minimization process we determined relies on the cache coherence state machines being Strongly Connected Components, without initial and terminal states.

   The abstraction algorithms make use of the fact that the state machines have stable states, interleaved with transitional states.

3. **Can programmatically implementable strategies be found when to apply which pattern?**

   The reduction algorithm can be fully automated. For the abstraction patterns the pattern application can be automated. The features to abstract or remove from the state machine must be identified by inspecting the state machine implementations to minimize.

4. **Can limits be found in the application of the minimization process?**

   We did not find any limits or restrictions to apply the abstraction patterns or reduction algorithms.

5. **How can the minimization process be best documented?**
The reduction algorithm is mathematically described and proven. The abstraction patterns are provided as named recipes, which define how to obtain the features that can be removed from the state machines.

6. Do the patterns have an ordering, is the use of certain patterns enabled by the application of other reduction patterns?

The abstraction patterns must be followed by the reduction algorithm. The reduction algorithm reduces the size of the state machine based on internal $r$-transitions and equal transitions that appear when features are removed by the abstraction patterns. The semantically preserving reduction patterns are therefore enabled by the abstraction patterns.

7. If the patterns have an ordering, can an optimal ordering be applied to obtain the best result?

The abstraction patterns we have found can be applied in any order, as they all target different types of features. The reduction algorithms will only provide meaningful results when applied after the abstraction patterns, as these rely on the removal of semantics by the abstraction patterns.

The main question that this research is trying to answer is the following:

*Is it possible to create sound abstractions of selected gem5 cache coherence state machines by programmatically applying state machine minimization patterns?*

We have found that we can create minimized state machines for the MI and MESI protocols. We have found no reasons to assume that we can not apply the same patterns and algorithms for other and more complex cache coherence protocols. Therefore we conclude that it is indeed possible to create sound abstractions by automatically applying the abstraction patterns and reduction algorithms we found.

### 9.3. Future work

We have applied the abstraction patterns and reduction algorithms to the MI and MESI protocols. Other protocols could also be targeted. This could further validate the applicability of the abstraction algorithms. It could be that analysing more complex protocols allows for the identification of other abstraction patterns that do not express themselves in the already investigated cache coherence state machines.

With regard to the semantically preserving reduction patterns, there could be algorithms that provide more efficient solutions. Future research could focus on finding these more efficient algorithms, e.g. weak- or branching bisimilarity reduction algorithms.

We have focussed on the creation of abstractions to research network impact of cache coherence protocols. Other research could be targeted at abstractions for other aspects of the cache coherence protocols, e.g. memory interfaces, the core interface or private data cache configurations. Our results should also be useable for these research directions, as
the abstraction patterns and reduction algorithms are not limited to network interactions. Creating abstractions for these other cache coherence features could also lead to the identification of additional abstraction patterns.

The reduction algorithm we devised is useable for all state machines which are SCCs and which have no initial and terminal states. Other research areas where these types of state machines are used could benefit from this research.
This appendix describes the information obtained and used during the analysis of the cache coherence protocols. It describes the various components that are used in the protocol, the queues that are used to communicate between these components, and the actions that are executed on state transitions. More information on these protocols can be obtained from the gem5 website, the URL to the protocol specific information is also provided. More information on the terminology can be found in [Sor+02].

A.1. MI PROTOCOL
The MI protocol is the simplest gem5 protocol, mainly used for educational purposes. The protocol is implemented for three types of components; cache controllers, each connected to a processing core; directory controllers, connected to memory; and DMA controllers, connected to IO related devices. These component types are detailed in the next sections.

A.1.1. CACHE CONTROLLER
The cache controller receives memory requests from cores and retrieves the associated memory data block from the directory controllers. When other cores or a DMA controller requires the memory block, it is relinquished again. A block can also be relinquished when the core requires the cache location for other purposes.

Queues
The input- and output queues are defined in the sourcecode with respectively the in_port and out_port functions. The following queues are defined for the cache controller:

1. forwardRequestNetwork_in: used to receive invalidates, getx requests, and writeback responses from the directory controller. These requests originate from other components.
2. responseNetwork_in: used to receive DATA packets over the network from the cache coherence components.
3. mandatoryQueue_in: input queue for core memory requests from the core for LOAD, STORE and IFetch requests.
4. requestNetwork_out: used to issue GETX and PUTX requests to the directory controller.
5. responseNetwork_out; used to send DATA packets to the other cache coherence components.

The queues that we are interested in are all the above queues, apart from the mandatoryQueue_in queue. That queue is used to receive requests from the processor core.

**STATES**

The description of all states in this protocol can be found at: [http://www.m5sim.org/MI_example](http://www.m5sim.org/MI_example)

**ACTIONS**

Table A.1: MI_example cache controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_issueRequest</td>
<td>Yes</td>
<td>Issues a GETX to the directory controller over the requestNetwork_out queue.</td>
</tr>
<tr>
<td>b_issuePUT</td>
<td>Yes</td>
<td>Issues a PUT request to the directory controller over the requestNetwork_out queue.</td>
</tr>
<tr>
<td>e_sendData</td>
<td>Yes</td>
<td>Send data from the cache to the requestor over the responseNetwork_out queue.</td>
</tr>
<tr>
<td>ee_sendDataFromTBE</td>
<td>Yes</td>
<td>Send data from the TBE buffers to the requestor over the responseNetwork_out queue.</td>
</tr>
<tr>
<td>n_popResponseQueue</td>
<td>Yes</td>
<td>Pop the responseNetwork_in queue.</td>
</tr>
<tr>
<td>o_popForwardedRequest Queue</td>
<td>Yes</td>
<td>Pop the forwardRequestNetwork_in queue.</td>
</tr>
<tr>
<td>i.allocateL1CacheBlock</td>
<td>No</td>
<td>Allocate a cache block.</td>
</tr>
<tr>
<td>h.deallocateL1CacheBlock</td>
<td>No</td>
<td>Deallocate a cache block.</td>
</tr>
<tr>
<td>m.popMandatoryQueue</td>
<td>No</td>
<td>pop the mandatory request queue mandatoryQueue_in.</td>
</tr>
<tr>
<td>p.profileMiss</td>
<td>No</td>
<td>update gem5 statistics.</td>
</tr>
<tr>
<td>p.profileHit</td>
<td>No</td>
<td>update gem5 statistics.</td>
</tr>
<tr>
<td>r.load_hit</td>
<td>No</td>
<td>Notify sequencer the load completed.</td>
</tr>
<tr>
<td>rx.load_hit</td>
<td>No</td>
<td>Notify the sequencer an external load completed.</td>
</tr>
<tr>
<td>s.store_hit</td>
<td>No</td>
<td>Notify the sequencer a store completed.</td>
</tr>
<tr>
<td>sx.store_hit</td>
<td>No</td>
<td>Notify the sequencer an external store completed.</td>
</tr>
<tr>
<td>u.writeDataToCache</td>
<td>No</td>
<td>Write data to cache.</td>
</tr>
<tr>
<td>forward_eviction_to_cpu</td>
<td>No</td>
<td>Send eviction information to the processor.</td>
</tr>
<tr>
<td>v.allocateTBE</td>
<td>No</td>
<td>Allocate TBE buffer.</td>
</tr>
<tr>
<td>w.deallocateTBE</td>
<td>No</td>
<td>Deallocate TBE buffer.</td>
</tr>
<tr>
<td>x.copyDataFromCacheToTBE</td>
<td>No</td>
<td>Copy data from cache to TBE.</td>
</tr>
<tr>
<td>z.stall</td>
<td>No</td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>
The actions for this component are described in Table A.1. The *impacted* column indicates whether the action is network-impacting (‘Yes’) or not network-impacting (‘No’).

### A.1.2. Directory Controller

The directory controller manages the communication with the memory subsystem. It also handles DMA write and read requests from the DMA component.

#### QUEUES

1. `dmaRequestQueue_in`: receive DMA READ and WRITE requests from the DMA controller.
2. `requestQueue_in`: incoming GETS, GETX and PUTX requests from cache controllers.
3. `memQueue_in`: incoming queue for memory read and writeback messages.
4. `forwardNetwork_out`: used to send INV, writeback ACKs, NACKs, and forward requests to cache controllers.
5. `responseNetwork_out`: used to send DATA packets to the other cache coherence components
6. `requestQueue_out`: does not seem to be used, queue definition is annotated with ‘For recycling requests’.
7. `dmaResponseNetwork_out`: used to send DATA and ACK messages.

#### STATES

The description of all states in this protocol can be found at: [http://www.m5sim.org/MI_example](http://www.m5sim.org/MI_example)

#### ACTIONS

Table A.2: MI_example directory controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_sendWriteBackAck</code></td>
<td>Yes</td>
<td>Send writeback ack to requestor over the <code>forwardNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>l_sendWriteBackAck</code></td>
<td>Yes</td>
<td>Send writeback ack to requestor over the <code>forwardNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>b_sendWriteBackNack</code></td>
<td>Yes</td>
<td>Send writeback nack to requestor over the <code>forwardNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>f_forwardRequest</code></td>
<td>Yes</td>
<td>Forward the request from the <code>requestQueue_in</code> queue to the <code>forwardNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>i_popIncomingRequestQueue</code></td>
<td>Yes</td>
<td>Pop the incoming request queue.</td>
</tr>
<tr>
<td><code>d_sendData</code></td>
<td>Yes</td>
<td>Send data to the requestor over the <code>responseNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>inv_sendCacheInvalidate</code></td>
<td>DMA</td>
<td>Invalidate a cache block for DMA by sending an INV message over the <code>forwardNetwork_out</code> queue.</td>
</tr>
<tr>
<td><code>dr_sendDMAData</code></td>
<td>DMA</td>
<td>Send data to DMA controller from directory over the <code>dmaResponseNetwork_out</code> queue.</td>
</tr>
</tbody>
</table>
The actions for this protocol are given in Table A.2. The `impacted` column indicates whether the action is network-impacting ('Yes'), used for cache-line ownership ('Share'), only network-impacting when DMA operations are taken into account ('DMA') or not network-impacting ('No').

### A.1.3. DMA CONTROLLER

#### QUEUES
1. `requestToDir_out`: send READ and WRITE requests to the directory.
2. `dmaRequestQueue_in`: input queue for LOAD and STORE requests from the DMA subsystem.
3. `dmaResponseQueue_in`: DATA and ACK messages from the directory controller.

#### STATES
The description of all states in this protocol can be found at: [http://www.m5sim.org/MI_example](http://www.m5sim.org/MI_example)
A.2. MESI PROTOCOL

The actions for this protocol are given in Table A.3. The impacted column indicates whether the action is not network-impacting (’No’) or only network-impacting when DMA operations are taken into account (’DMA’).

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_sendReadRequest</td>
<td>DMA</td>
<td>Send a DMA read request to memory over the requestToDir_out queue.</td>
</tr>
<tr>
<td>s_sendWriteRequest</td>
<td>DMA</td>
<td>Send a DMA write request to memory over the requestToDir_out queue.</td>
</tr>
<tr>
<td>p_popRequestQueue</td>
<td>DMA</td>
<td>Pop request queue dmaRequestQueue_in.</td>
</tr>
<tr>
<td>p_popResponseQueue</td>
<td>DMA</td>
<td>Pop response queue dmaResponseQueue_in.</td>
</tr>
<tr>
<td>a_ackCallback</td>
<td>No</td>
<td>Notify dma controller that write request completed.</td>
</tr>
<tr>
<td>d_dataCallback</td>
<td>No</td>
<td>Write data to dma sequencer.</td>
</tr>
</tbody>
</table>

A.1.4. MORE INFORMATION

More information on the MI_example gem5 protocol can be obtained from the following locations:

- **Description** - http://www.m5sim.org/MI_example
- **SLICC description** - http://www.m5sim.org/SLICC
- **Cache code** - http://grok.gem5.org/source/xref/gem5/src/mem/protocol/MI_example-cache.sm

For the source listings the latest history version before August 26, 2014 should be taken to validate the above data.

A.2. MESI PROTOCOL

The MESI protocol

The gem5 simulator (gem5) MESI protocol we investigated is a two level protocol with for each core a private L1 cache, with separate data and instruction caches. The L2 cache is a second level cache which is shared among the cores.
The protocol has four stable states, Modified means that the cache-line is written, E means that the cache-line has exclusive permission to be written but is still clean, S is a shared read-only copy, I means that the cache-line is invalid.

The protocol is implemented with four types of components; L1 cache controllers, each connected to a processing core; L2 cache controllers for the shared cache; directory controllers, connected to memory; and DMA controllers, connected to IO related devices. These four component types are detailed in the next sections.

A.2.1. L1 CACHE CONTROLLER
The L1 cache controller receives memory requests from cores and retrieves the associated cache-line from the L2 cache controllers. When other cores or a DMA controller requires the cache-line, it is relinquished again. A cache-line can also be relinquished in a replacement, when the L1 cache wants to replace the cache-line content for another memory block.

Queues
The input- and output queues are defined in the source code with respectively the in_port and out_port functions. The following queues are defined for the cache controller:

1. optionalQueue_in: used by the core prefetcher to send pre-fetch cache-line instructions.
2. optionalQueue_out: used by the core prefetcher to send pre-fetch cache-line instructions.
3. mandatoryQueue_in: input queue for core memory requests from the core for LOAD, STORE and IFetch requests.
4. requestL1Network_out*: used to issue requests to the other nodes.
5. requestL1Network_in*: requests from this L1 cache to the shared L2
6. responseL1Network_in*: used to receive DATA packets over the network from the cache coherence components.
7. responseL1Network_out*: used to send DATA packets to the other cache coherence components.
8. unblockNetwork_out*: used to send unblock requests to the L2 cache controller.

The queues that we are interested in are all the above queues marked with a *. The other queues are used to receive requests from the processor core.

States
The description of all states in this protocol can be found at: http://www.m5sim.org/MESI_Two_Level

Actions

Table A.4: MESI L1 cache controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_issueGETS</td>
<td>Yes</td>
<td>Issues a GETS over the requestL1Network_out queue</td>
</tr>
<tr>
<td>Action name</td>
<td>impacting</td>
<td>description (continued)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>pa_issuePfGETS</td>
<td>Yes</td>
<td>Issues a prefetch GETS over the requestL1Network_out queue</td>
</tr>
<tr>
<td>ai_issueGETINSTR</td>
<td>Yes</td>
<td>Issue GETINSTR over the requestL1Network_out queue</td>
</tr>
<tr>
<td>pai_issuePfGETSINSTR</td>
<td>Yes</td>
<td>Issue GETINSTR for Prefetch request over the requestL1Network_out queue</td>
</tr>
<tr>
<td>b_issueGETX</td>
<td>Yes</td>
<td>Issue GETX over the requestL1Network_out queue</td>
</tr>
<tr>
<td>pb_issuePfGETX</td>
<td>Yes</td>
<td>Issue prefetch GETX over the requestL1Network_out queue</td>
</tr>
<tr>
<td>c_issueUPGRADE</td>
<td>Yes</td>
<td>Issue an upgrade message over the requestL1Network_out queue</td>
</tr>
<tr>
<td>d_sendDataToRequestor</td>
<td>Yes</td>
<td>send data to requestor over the responseL1Network_out queue</td>
</tr>
<tr>
<td>d2_sendDataToL2</td>
<td>Yes</td>
<td>Send data to the L2 cache because of M downgrade over the responseL1Network_out queue</td>
</tr>
<tr>
<td>dt_sendDataToRequestor</td>
<td>Yes</td>
<td>send data to requestor over the responseL1Network_out queue</td>
</tr>
<tr>
<td>d2t_sendDataToL2_fromTBE</td>
<td>Yes</td>
<td>send data to the L2 cache over the responseL1Network_out queue</td>
</tr>
<tr>
<td>e_sendAckToRequestor</td>
<td>Yes</td>
<td>send invalidate ack to requestor (could be L2 or L1) over the responseL1Network_out queue</td>
</tr>
<tr>
<td>f_sendDataToL2</td>
<td>Yes</td>
<td>send data to the L2 cache over the responseL1Network_out queue</td>
</tr>
<tr>
<td>ft_sendDataToL2_fromTBE</td>
<td>Yes</td>
<td>send data to the L2 cache from TBE over the responseL1Network_out queue</td>
</tr>
<tr>
<td>fi_sendInvAck</td>
<td>Yes</td>
<td>send Acknowledge to the L2 over the responseL1Network_out queue</td>
</tr>
<tr>
<td>forward_eviction_to_cpu</td>
<td>No</td>
<td>sends eviction information to the processor</td>
</tr>
<tr>
<td>g_issuePUTX</td>
<td>Yes</td>
<td>Send data to the L2 cache over the requestL1Network_out queue</td>
</tr>
<tr>
<td>j_sendUnblock</td>
<td>Yes</td>
<td>Send unblock to the L2 cache over the unblockNetwork_out queue</td>
</tr>
<tr>
<td>jj_sendUnblock</td>
<td>Yes</td>
<td>Send exclusive unblock to the L2 cache over the unblockNetwork_out queue</td>
</tr>
<tr>
<td>dg_invalidate_sc</td>
<td>No</td>
<td>Invalidate store conditional as the cache lost permission</td>
</tr>
<tr>
<td>h_load_hit</td>
<td>No</td>
<td>if not prefetch, notify sequencer the load completed</td>
</tr>
<tr>
<td>hx_load_hit</td>
<td>No</td>
<td>if not prefetch, notify sequencer the load completed</td>
</tr>
<tr>
<td>hh_store_hit</td>
<td>No</td>
<td>if not prefetch, notify sequencer the store completed</td>
</tr>
</tbody>
</table>
### A. Cache Coherence Protocol Details

#### Action name | impacting | description (continued)
--- | --- | ---
hhx_store_hit | No | if not prefetch, notify sequencer the store completed
i_allocateTBE | No | allocate TBE buffer
k_popMandatoryQueue | No | pop the mandatory queue
l_popRequestQueue | Yes | pop the requestL1network_in queue
o_popIncomingResponse | Yes | pop the responseL1Network_in queue
s_deallocateTBE | No | Deallocate the TBE
u_writeDataToL1Cache | No | Write data to cache
q_updateAckCount | No | Update the ack count
ff_deallocateL1CacheBlock | No | Deallocate L1 cache block
oo_allocateL1DCacheBlock | No | Set L1 D-cache tag equal to tag of block B
pp_allocateL1ICacheBlock | No | Set L1 I-cache tag equal to tag of block B
z_stallAndWaitMandatory | No | recycle mandatory queue
kd_wakeUpDependents | No | wake-up dependents
uu_profileInstMiss | No | Profile the demand miss
uu_profileInstHit | No | Profile the demand hit
uu_profileDataMiss | No | Profile the demand miss
uu_profileDataHit | No | Profile the demand hit
po_observeMiss | No | Inform the prefetcher about the miss
ppm_observePfMiss | No | Inform the prefetcher about the partial miss
pq_popPrefetchQueue | No | pop the optionalQueue_in queue
mp_markPrefetched | No | Mark the cache_entry (cache-line) as pre-fetched

The actions for this component are described in Table A.1. The *impacted* column indicates whether the action is network-impacting ('Yes') or not network-impacting ('No').

#### A.2.2. L2 Cache Controller

The L2 cache controller receives memory requests L1 caches and retrieves the associated memory cache-line from memory. The Directory controller can request to relinquish a cache-line to be used by a DMA controller.

#### Queues

The input- and output queues are defined in the sourcecode with respectively the `in_port` and `out_port` functions. The following queues are defined for the cache controller:

1. L1RequestL2Network_out: L1 Requests from L2 cache
2. L1unblockNetwork_in: Unblock requests from L1 caches
3. L1RequestL2Network_in: L1 cache requests to L2 cache
4. DirRequestL2Network_out: Directory requests from L2 cache
5. responseL2Network_out: Responses from L2 cache to Directory and L1 caches
6. responseL2Network_in: Responses from Directory and L1 caches to L2 cache
Which queues are network-impacting is dependent on the network configuration. The first three queues are related to the L1 network, between the L1 caches and the L2 cache.

The last three queues in the above list are related to the L2 network, between the L2 cache, the Directory controller and the DMA controller(s).

**STATES**

The description of all states in this protocol can be found at: [http://www.m5sim.org/MESI_Two_Level](http://www.m5sim.org/MESI_Two_Level)

**ACTIONS**

Table A.5: MESI L2 cache controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_issueFetchToMemory</td>
<td>Yes</td>
<td>fetch data (GETS) from memory over the DirRequestL2Network_out queue</td>
</tr>
<tr>
<td>b_forwardRequestToExclusive</td>
<td>Yes</td>
<td>Forward request to the exclusive L1 over the L1RequestL2Network_out</td>
</tr>
<tr>
<td>c_exclusiveReplacement</td>
<td>Yes</td>
<td>Send MEMORY_DATA to memory over the responseL2Network_out</td>
</tr>
<tr>
<td>c_exclusiveCleanReplacement</td>
<td>Yes</td>
<td>Send ACK to memory for clean replacement over the responseL2Network_out</td>
</tr>
<tr>
<td>ct_exclusiveReplacement</td>
<td>Yes</td>
<td>Send MEMORY_DATA to memory over responseL2Network_out</td>
</tr>
<tr>
<td>d_sendDataToRequestor</td>
<td>Yes</td>
<td>Send DATA from cache to requestor over responseL2Network_out</td>
</tr>
<tr>
<td>dd_sendExclusiveDataToRequestor</td>
<td>Yes</td>
<td>Send DATA_EXCLUSIVE from cache to requestor over responseL2Network_out</td>
</tr>
<tr>
<td>ds_sendSharedDataToRequestor</td>
<td>Yes</td>
<td>Send DATA from cache to requestor over responseL2Network_out</td>
</tr>
<tr>
<td>e_sendDataToGetSRequestors</td>
<td>Yes</td>
<td>Send DATA from cache to all GetS IDs over responseL2Network_out</td>
</tr>
<tr>
<td>ex_sendExclusiveDataToGetSRequestors</td>
<td>Yes</td>
<td>Send DATA_EXCLUSIVE from cache to all GetS IDs</td>
</tr>
<tr>
<td>ee_sendDataToGetXRequestor</td>
<td>Yes</td>
<td>Send DATA from cache to GetX ID over responseL2Network_out</td>
</tr>
<tr>
<td>f_sendInvToSharers</td>
<td>Yes</td>
<td>Send INV-alidate sharers for L2 replacement over L1RequestL2Network_out</td>
</tr>
<tr>
<td>fw_sendFwdInvToSharers</td>
<td>Yes</td>
<td>Send INV-alidate sharers for request, over L1RequestL2Network_out</td>
</tr>
<tr>
<td>fwm_sendFwdInvToSharers</td>
<td>Yes</td>
<td>Send INV-alidate sharers for request, requestor is sharer, over</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1RequestL2Network_out</td>
</tr>
<tr>
<td>i_allocateTBE</td>
<td>No</td>
<td>Allocate TBE for request</td>
</tr>
<tr>
<td>s_deallocateTBE</td>
<td>No</td>
<td>Deallocate external TBE</td>
</tr>
<tr>
<td>jj_popL1RequestQueue</td>
<td>Yes</td>
<td>Pop incoming L1 request</td>
</tr>
</tbody>
</table>
The actions for this component are described in Table A.5. The impacted column indicates whether the action is network-impacting (‘Yes’) or not network-impacting (‘No’). The entries marked with ‘Shared’ are not network-impacting in itself, but are needed to maintain the cache-line ownership state.
A.2.3. **DIRECTORY CONTROLLER**

The Directory controller receives memory requests and stores from the L2 cache and the DMA controller. When the DMA controller requests a memory block the directory controller can request the L2 cache to relinquish the cache-line.

**Queues**

The input- and output queues are defined in the sourcecode with respectively the `in_port` and `out_port` functions. The following queues are defined for the cache controller:

1. `responseNetwork_in`: Memory DATA or ACK’s from the L2 caches
2. `responseNetwork_out`: Responses from Directory to the L2 caches
3. `requestNetwork_in`: Requests from L2 caches or DMA nodes
4. `memQueue_out`: Requests to memory
5. `memQueue_in`: DATA or ACK messages from memory.

The queues that we are interested in are the top three queues from the list above. The memQueues are used for communication with the memory subsection.

The `responseNetwork_out` queue is also used for INV requests to the L2 cache, in the `inv_sendCacheInvalidate` action.

**States**

The description of all states in this protocol can be found at: [http://www.m5sim.org/MESI_Two_Level](http://www.m5sim.org/MESI_Two_Level)

**Actions**

Table A.6: MESI Directory controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_sendAck</td>
<td>Yes</td>
<td>send a MEMORY_ACK to L2 over the <code>responseNetwork_out</code> (from <code>responseNetwork_in</code>, back to sender)</td>
</tr>
<tr>
<td>d_sendData</td>
<td>Yes</td>
<td>send MEMORY_DATA to L2 over the <code>responseNetwork_out</code></td>
</tr>
<tr>
<td>aa_sendAck</td>
<td>Yes</td>
<td>send a MEMORY_ACK to L2 over the <code>responseNetwork_out</code> (from <code>memQueue_in</code>)</td>
</tr>
<tr>
<td>j_popIncomingRequestQueue</td>
<td>Yes</td>
<td>Pop incoming request queue <code>requestNetwork_in</code></td>
</tr>
<tr>
<td>k_popIncomingResponseQueue</td>
<td>Yes</td>
<td>Pop incoming response queue <code>responseNetwork_in</code></td>
</tr>
<tr>
<td>l_popMemQueue</td>
<td>No</td>
<td>Pop off-chip request queue <code>memQueue_in</code></td>
</tr>
<tr>
<td>kd_wakeUpDependents</td>
<td>No</td>
<td>wake-up dependents</td>
</tr>
<tr>
<td>qf_queueMemoryFetchRequest</td>
<td>No</td>
<td>Queue off-chip fetch request using <code>memQueue_out</code></td>
</tr>
<tr>
<td>qw_queueMemoryWBRequest</td>
<td>No</td>
<td>Queue off-chip writeback request using <code>memQueue_out</code></td>
</tr>
</tbody>
</table>
## A. Cache Coherence Protocol Details

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_writeDataToMemory</td>
<td>No</td>
<td>Write dirty writeback to memory</td>
</tr>
<tr>
<td>qf_queueMemoryFetchRequestDMA</td>
<td>No</td>
<td>Queue off-chip fetch request using memQueue_out</td>
</tr>
<tr>
<td>p_popIncomingDMARequestQueue</td>
<td>DMA</td>
<td>Pop incoming DMA queue from requestNetwork_in</td>
</tr>
<tr>
<td>dr_sendDMAData</td>
<td>DMA</td>
<td>Send Data to DMA controller from directory over the responseNetwork_out</td>
</tr>
<tr>
<td>dw_writeDMAData</td>
<td>No</td>
<td>DMA write data to memory</td>
</tr>
<tr>
<td>qw_queueMemoryWBRequest_partial</td>
<td>No</td>
<td>Queue off-chip writeback request using memQueue_out</td>
</tr>
<tr>
<td>da_sendDMAAck</td>
<td>DMA</td>
<td>Send Ack to DMA controller over the responseNetwork_out queue</td>
</tr>
<tr>
<td>z_stallAndWaitRequest</td>
<td>No</td>
<td>recycle request queue requestNetwork_in</td>
</tr>
<tr>
<td>zz_recycleDMAQueue</td>
<td>No</td>
<td>recycle DMA queue requestNetwork_in</td>
</tr>
<tr>
<td>inv_sendCacheInvalidate</td>
<td>DMA</td>
<td>Invalidate a cache block to responseNetwork_out</td>
</tr>
<tr>
<td>drp_sendDMAData</td>
<td>DMA</td>
<td>Send Data to DMA controller from incoming PUTX over responseNetwork_out</td>
</tr>
<tr>
<td>v_allocateTBE</td>
<td>No</td>
<td>Allocate TBE</td>
</tr>
<tr>
<td>dwt_writeDMADataFromTBE</td>
<td>No</td>
<td>DMA Write data to memory from TBE</td>
</tr>
<tr>
<td>qw_queueMemoryWBRequest_noTBE</td>
<td>No</td>
<td>Queue off-chip writeback request using memQueue_out</td>
</tr>
<tr>
<td>w_deallocateTBE</td>
<td>No</td>
<td>Deallocate TBE</td>
</tr>
</tbody>
</table>

The actions for this component are described in Table A.6. The *impacted* column indicates whether the action is network-impacting (‘Yes’), network-impacting for DMA actions (‘DMA’) or not network-impacting (‘No’).

### A.2.4. DMA Controller

The DMA controller initiates memory requests for block read- and write actions. It targets these requests at the directory controller.

#### Queues

The input- and output queues are defined in the source code with respectively the `in_port` and `out_port` functions. The following queues are defined for the cache controller:

1. `requestToDir_out`: DMA requests to the Directory controller
2. `dmaRequestQueue_in`: DMA requests from the IO device.
3. `dmaResponseQueue_in`: Responses from L2 cache

The `requestToDir_out` queue and the `dmaResponseQueue_in` are the queues that we are interested in. The `dmaRequestQueue_in` is non-network-impacting.
A.2. MESI PROTOCOL

STATES
There are three states in this state machine:

1. READY: Invalid, Ready to accept a new request;
2. BUSY_RD: Busy: currently processing a read request;
3. BUSY_WR: Busy: currently processing a write request;

ACTIONS

Table A.7: MESI L2 cache controller actions

<table>
<thead>
<tr>
<th>Action name</th>
<th>impacting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_sendReadRequest</td>
<td>DMA</td>
<td>Send a DMA READ request over the requestToDir_out queue</td>
</tr>
<tr>
<td>s_sendWriteRequest</td>
<td>DMA</td>
<td>Send a DMA WRITE request over the requestToDir_out queue</td>
</tr>
<tr>
<td>a_ackCallback</td>
<td>No</td>
<td>Notify dma controller that write request completed</td>
</tr>
<tr>
<td>d_dataCallback</td>
<td>No</td>
<td>Writer data to dma sequencer</td>
</tr>
<tr>
<td>p_popRequestQueue</td>
<td>No</td>
<td>pop dmaRequestQueue_in request queue</td>
</tr>
<tr>
<td>p_popResponseQueue</td>
<td>DMA</td>
<td>Pop dmaResponseQueue_in response queue</td>
</tr>
</tbody>
</table>

The actions for this component are described in Table A.7. The impacted column indicates whether the action is network-impacting for DMA actions ('DMA') or not network-impacting ('No').

A.2.5. MORE INFORMATION
More information on the MESI gem5 protocol can be obtained from the following locations:

- Description - http://www.m5sim.org/MESI_Two_Level
- SLICC description - http://www.m5sim.org/SLICC

For the source listings the latest history version before August 26, 2014 should be taken to validate the above data.
BIBLIOGRAPHY

BOOKS


SCIENTIFIC ARTICLES


Ọụzọ 1:79

**CONFERENCES**


**TECHNICAL DOCUMENTATION**


ACRONYMS

**CPU** Central Processing Unit.

**DFA** Deterministic Finite Automaton.

**LTS** Labeled Transition System.

**NFA** Non-deterministic finite automaton.

**NoC** Network on a Chip.

**SCC** Strongly Connected Component.

**SE** Software Engineering.
**GLOSSARY**

**AP-Deterministic** a deterministic state machine where all states are labeled with the properties that are valid in that state. AP stands for Atomic Propositions or Properties. Deterministic means that the state machine only allows at most a single state to transition to with a certain set of properties. Also known as deterministic Kripke structures.

**bijection** in mathematics, a bijection, bijective function or one-to-one correspondence is a function between the elements of two sets, where every element of one set is paired with exactly one element of the other set, and every element of the other set is paired with exactly one element of the first set.

**bisimilarity** the finest extensional behavioural equivalence one would like to impose on processes.

**cache coherence** cache communication protocols with which to make the caches of a shared-memory system as functionally invisible as caches in a single core system.

**cache controller** a module that services load and store requests for cached values in the cache. See [SHW11, p. 83].

**cache-line** a block of memory, normally in the order of 64 or 128 bytes, that is handled as a single entity in the system caches. This is normally also equal to the size of a memory request from and to main memory.

**core** a microprocessor execution unit capable of actively execute application actions. In the context of this paper these are the units initiating memory load- and store actions.

**deadlock** a situation where two or more actions are waiting for each other, caused by a circular dependency. These situations typically never resolve without external interference.

**Deterministic Finite Automaton** state machines that allow at most a single transition for an event. No choices need to be made, and it is always clear what the next state will be.

**die** a small block of semiconductor material, on which an integrated circuit is fabricated. It is fabricated in large batches on a single wafer. A microprocessor or memory chip is normally composed of a single die placed in a 'chip' package.

**directory controller** a module that services load and store requests for main memory content. It can also maintain the ownership of the cache-lines. If it does not, the component is sometimes also called a memory controller. See [SHW11, p. 84].
**extensional property**  a property whose definition only takes into account the interactions that the processes may, or may not, perform [San12, p. 2].

**gem5**  the gem5 simulator is an open source software application for simulation of computer systems in software. See: http://gem5.org.

**Labeled Transition System**  a state machine where the transitions are labeled. The labels model the actions that are performed during the transition, the states model the system states.

**memory consistency**  ruleset defining the allowed behavior of multithreaded programs executing with shared memory.

**multi-threading**  for software: the ability of a single process or application to subdivide its work in multiple threads of execution, which can be run in parallel; for microprocessor cores: the ability to concurrently run multiple software threads in parallel on a single core.

**network on a chip**  a communication subsystem between various execution cores and supporting subsystems on an integrated circuit.

**Non-deterministic finite automaton**  state machines that allow multiple transition for an event, and internal transitions, taken without external stimuli. Potentially choices need to be made, and a trace through an NFA can use multiple paths.

**state machine**  a finite-state machine, or a state machine, is an abstract machine that describes a number of states, and the events and triggers with which the machine transitions between states.

**state space explosion**  a combinatorial explosion or blowup of the number of states that must be examined to get a state machine property validated with automatic analysis.

**Strongly Connected Component**  part of a state machine where all states are strongly connected, i.e. where the states can transition to one another, via a path in both directions between each individual state.

**structural equivalence**  two state machines are structurally equivalent or isomorphic when a bijection can be established on the states and their transitions. See also [San12, p. 16].

**trace equivalence**  two state machines are trace equivalent when they can perform the same finite sequences of transactions. See also [San12, p. 18].

**wormhole routing**  a simple flow control method where a packet is split into a number of short fixed messages called 'flit's. The first flit contains the address and with it creates the connection for the following flit's, which follow the head.