Towards the Automatic Synthesis of Cache Coherence Protocols

Theo Olausson

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

This paper presents extensive contributions towards the automatic synthesis of cache coherence protocols. Motivated by the insight that even the most sophisticated current approaches still assume the existence of a correct *Stable State Protocol* (SSP), an abstract specification of the protocol under atomic conditions, it targets these abstractions in particular. It lays the groundwork for future research towards synthesis of SSPs by suggesting a concrete correctness criterion for SSPs, presenting the first publicly available dataset on which bug localization and synthesis efforts can be compared, and deriving a taxonomy of the bugs which appear in cache coherence protocols. Finally, two novel heuristic methods for the localisation of bugs in SSPs are presented and evaluated. The results of this evaluation indicate that the two methods can often identify bugs in protocols at a finer granularity than is currently possible with standard model checking engines. Thus, the methods may serve as useful debugging tools when developing cache coherence protocols, and open up for future work towards synthesising fixes to the bugs.

Acknowledgements

First of all, I would like to thank Dr. Vijay Nagarajan. Thank you for supervising this project, offering your expertise and guidance. More importantly, thank you for helping me discover and nurture my interest for scientific research, and for being a great mentor to me these past few years.

Secondly, I want to extend a sincere ’thank you’ to all the close friends and helpful colleagues I have had the privilege of getting to know since coming to the UK. Without you, my life in this home away from home would never have been the same.

Finally, I would like to thank my family and loved ones for their unending patience and support. Nothing that I have done in my life would I have been able to do without you.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Cache coherence protocols are notoriously difficult to implement, debug, and maintain.

– Chandra et al. (1997)

... [H]ardware cache coherence protocols still remain difficult to design, understand, and prove correct.

– Komuravelli et al. (2014)

... [D]esigning and verifying a new hardware coherence protocol is difficult...

– Alsop et al. (2018)

Since first appearing in the late 1970s, hardware cache coherence protocols have served a critical role in both commercial and academic multiprocessor systems. By keeping data consistent even when it is replicated among many private caches, they have helped make multiprocessor systems performant and easier to program for over 40 years. Unfortunately, since the very first time these protocols graced our academic articles and workplace whiteboards, they have also remained a constant thorn in the sides of the architects and engineers building those very same systems.

The problem with cache coherence protocols is twofold. First, they are difficult to reuse. Cache coherence protocols are unfortunately ”not off-the-shelf items, as their details depend on the requirements of the system under consideration” (Chandra et al., 1997). Secondly, their implementation and the verification thereof is a laborious and error-prone process. If the three quotes above were not convincing enough of this, I invite the reader to try it themselves with Carnegie Mellon University’s excellent online exercise (Gold, 2009).

As a result of this complexity, the last 20 or so years has seen an increasing interest within the academic community to make cache coherence protocols easier to develop. Using ideas from program synthesis and exploiting key insights about the nature of these protocols, recent works have been able to automate away the most back-breaking tasks, such as filling in holes in the protocols caused by race conditions on the interconnects between the processors. However, it still remains the case that the developer has to do a lot of heavy lifting themselves.

Today, commercial computer systems in any prince range are increasingly incorporating neural network accelerators, on-die graphics processing units, and dynamically scheduled pairs of low-power/high-performance cores. These specialized units enable the systems to handle modern workloads, but currently they remain difficult to program because their communication relies on manual passing of data between the units. Cache coherence protocols could alleviate this issue by presenting a unified, consistent view of the memory to each unit. Unfortunately, developing a new coherence protocol for any system is challenging enough, and when that system consists of several specialized units with different memory access patterns and that are perhaps even supplied by different vendors, the task becomes near insurmountable. If computing is to continue to enjoy the substantial growth in performance that it has seen for the last century, then overcoming this challenge is now more pressing than ever.

1.2 Project Goals and Contributions

The goal of this Master of Informatics project is to establish new methods for the pinpointing of, and synthesis of fixes to, bugs in cache coherence protocols. In the initial project proposal, the completion criteria were laid out as follows: *either a software which is capable of fixing the bugs in some set of coherence protocols, or significant experimental and theoretical results regarding the feasibility of such a task.*

This dissertation presents Part 1 of this project, which lays the groundwork for future research in this area. Specifically, it seeks to extend the work of ProtoGen (see section 2.4) by targeting the Stable State Protocols (see section 2.2) that it takes as input. Its main contributions are:

- A correctness criterion and verification procedure for Stable State Protocols (chapter 3);
- A micro-benchmark of buggy protocols, and a proposed taxonomy of the observed bugs (chapter 4);
- Two novel heuristic methods for the localisation of bugs in Stable State Protocols (chapter 5);
- A branch of ProtoGen implementing all of the above.

1.3 Outline of Report

We begin this report by discussing cache coherence and prior work which this dissertation builds upon in chapter 2. Then, chapters 3 through 5 will discuss the main contributions of this dissertation, in the order presented above. Having introduced each method, we devote chapter 6 to evaluating our work. Finally, chapter 7 summarises the ground we have covered and identifies future work in this area.

Chapter 2

Background

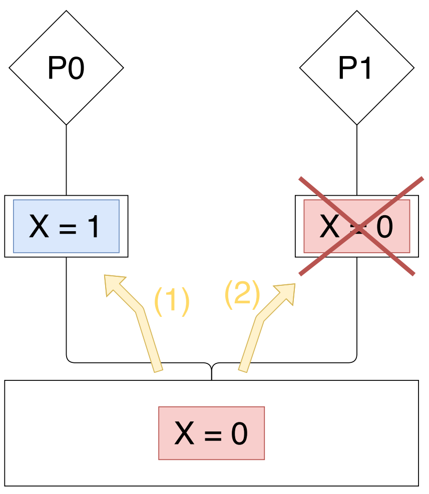
In this chapter, we look at the concepts and techniques which form the foundation of our work. We begin by giving an overview of coherence in section 2.1, introducing the problem and discussing how it may be resolved by using cache coherence protocols. We then take a more detailed look at cache coherence protocols in section 2.2, introducing the MSI, MESI, and MOSI protocols which will be frequently used throughout this dissertation. Having discussed these protocols at length, we next move on to introducing the model checker Murphi in section 2.3, before discussing ProtoGen in section 2.4. Finally, we round off the chapter in section 2.5 by briefly discussing other work which is related (but not essential) to that presented in this dissertation.

2.1 Coherence in Parallel Computer Architectures

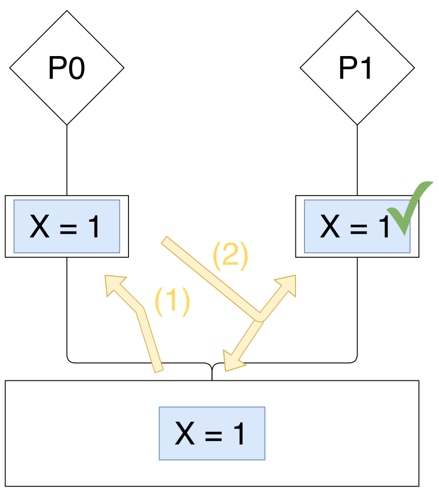
For a long time, computer systems enjoyed dramatic performance gains due to an exponential growth in transistor counts, an observation first made by Moore (1965). In the last few decades, this growth in transistor count has slowed down to a crawl. This has sparked a shift in computer architecture towards multi- or many-processor systems, where several processors work in parallel. To allow for efficient communication and co-operation within such multi-processor systems, the processors often share the entire memory space (i.e. both disks and main memory); such architectures are known as shared-memory architectures (Valduriez, 2009).

Shared-memory architectures, while efficient and easy to program, lead to complications. When a processor accesses a resource, it may store it in a local (private) cache to speed up future computation. If this access is a write, constructing correct parallel programs requires that the new value is propagated to the other processors. Similarly, if the access is a read but another processor later performs a write, the initial processor must be told that the value which it has in its local cache is stale and can no longer be trusted. Figure 2.1a shows one example scenario.

To resolve these issues, we need *coherence*. We will shortly define coherence formally, but for now one can think of it as a structured way of making sure hard-to-reason-about scenarios such as in figure 2.1a play out more nicely, as in figure 2.1b. Coherence can



(a) Without coherence: (1) P0 writes X=1, caching the result; (2) P1 reads X, retrieves the stale value X=0 from main memory!



(b) With coherence: (1) P0 writes X=1, caching the result; (2) P1 reads X, fetching the value from P0's cache and (optionally) causing it to be written back to main memory.

Figure 2.1: An example of how coherence affects the value returned by an external read following a write at a processor.

be provided by software, for example at the language or library level. More commonly, coherence is provided by hardware structures called cache coherence protocols, which are the focus of this dissertation.

Cache coherence protocols provide coherence in hardware by giving the caches a structured way of keeping track of and communicating changes to the state of memory locations. They can either be consistency-agnostic, meaning they will provide coherence regardless of what the *memory consistency model* is, or they can directly target the memory consistency model. In this dissertation, we will only consider consistency-agnostic protocols, and we will therefore not devote time to introducing the concept of a memory consistency model here. [1]

According to Nagarajan et al. (2020), there are two criteria a cache coherence protocol needs to meet in order to be consistency-agnostically coherent. The first is to maintain the Single-Writer-Multiple-Reader invariant:

> **Definition 2.1.** A shared-memory system upholds the **Single-Writer-Multiple-Reader (SWMR) invariant** if, and only if, for any given memory location, at any given moment in (logical) time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it.

The second is to uphold the so called data invariant:

> **Definition 2.2.** A shared-memory system upholds the **data invariant** if and only

[1] The keen reader may find the introductory chapter of Nagarajan et al. (2020) to be an accessible introduction to memory consistency models.

if for every memory location, the value of the memory location at the start of an epoch is always the same as the value of the same memory location in its last read-write epoch.

However, in this dissertation we will use a slightly tweaked definition of coherence:

Definition 2.3. A shared-memory system is **coherent** if and only if it maintains the SWMR invariant (definition 2.1) and is free of deadlocks.

In this alternative definition, we remove the second of Nagarajan et al. (2020)'s conditions while also making explicit that we want the system to be free of deadlocks. There are two motivations behind this change. The first is to make clear that the concept of a memory model (and therefore the values observed by reads) is not essential to the work laid out in this dissertation. The second is that this is a practical change, since the model-checking verification tools we use are not very well suited to verify this second criterion (but are good tools for checking SWMR and deadlocks); a more common practice is to verify the propagation of memory values by running so called litmus tests (Alglave et al., 2011). By removing the data invariant from consideration, we can thus focus our attention fully on verifying that our protocols satisfy SWMR and are free of deadlocks. For the rest of this dissertation, whenever we refer to *coherence* it is this definition which we are referring to.

Let us now look at some different types of cache coherence protocols in more detail.

2.2 Cache Coherence Protocols

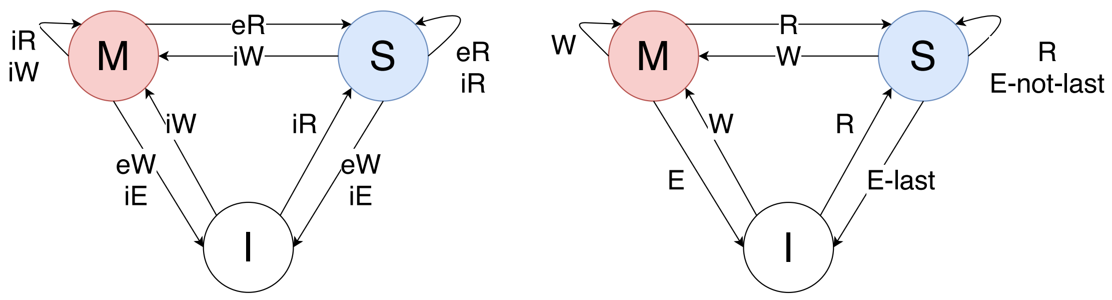
There are two main types of cache coherence protocols. *Snooping protocols* have a single interconnect spanning all of the processors and the memory controller, and communication is done in a broadcasted, undirected manner over this interconnect. *Directory protocols* take a more structured approach, maintaining a *directory* at the memory controller with state about each cache line, such as whether someone is currently accessing it. Instead of broadcasting their messages over a large, shared interconnect, directory protocols can therefore use simple peer-to-peer connections; the directory simply serves as a middleman between the processors, forwarding and replying to requests as needed to maintain coherence. Though directory protocols add some overhead both in terms of memory (storing the directory) and logical complexity (involving more diverse messages being sent between more actors), they scale better due to avoiding the need for broadcasting (Nagarajan et al., 2020).

Having settled on a messaging methodology, the specification of a cache coherence protocol comes down to describing its states and transitions. In textbooks, papers and presentations this is often done on a very abstract level, describing the protocols by pretending that each request is handled atomically. Such abstractions of the protocols are often called *Stable State Protocols* (SSPs), a term which we will make heave use of in this work. Unfortunately, this pretense quickly falls apart in the real world, because responding to a request typically requires co-ordination between several members of the memory system. During the time it takes for everyone to agree on what is happening, someone else might therefore very well also fire off a request. To handle this, cache coherence protocols need *transient states*: intermediary states which the machines take on whilst waiting for replies from others. Nonetheless this abstraction remains useful, as it allows writers and readers to quickly gain an understanding of the conceptual ideas of a protocol, leaving the details of the implementation until later.

2.2.1 Protocols Targeted in This Dissertation

In this work we focus exclusively on directory coherence protocols, as this is the class of protocols which ProtoGen targets. We will base our discussion around three common consistency-agnostic directory protocols: MSI, MOSI, and MESI.

MSI is a consistency-agnostic cache coherence protocol consisting of three stable states: **M**odified, **S**hared, and **I**nvalid. These states directly correlate to having read-and-write (also known as read-exclusive), read-only, and no permissions to the cache line. Figure 2.2 presents the transitions between theses states on a very abstract level, in terms of internal and external accesses to the cache line. It should be noted that this specification is *not* quite a complete SSP, because it does not define the communication between the directory and the cache controllers.



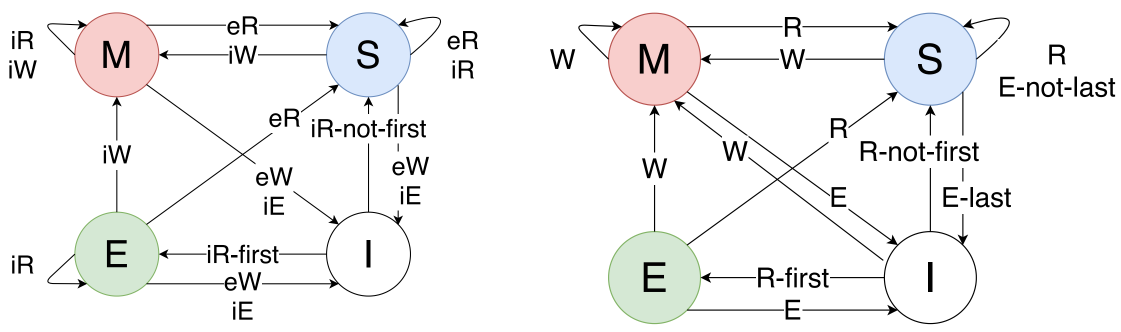
(a) MSI cache controller (b) MSI directory controller

Figure 2.2: MSI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine). Note that the directory needs to differentiate between when the last sharer evicts the line and when there are still other sharers holding the line.

MESI extends MSI with the **E**xclusive state. This state is used when a cache has only requested read permission, but is the first machine to do so; there are no other machines currently caching the line. Like the `S` state it gives the cache only read permission, but unlike the `S` state it also allows the machine to silently (i.e. without communicating with anyone else) upgrade itself to `M` state.[2] This optimization eliminates redundant upgrades in permission of private variables, and can thus speed up execution depending on the workload. Figure 2.3 gives the abstract specification of the MESI protocol.

MOSI instead extends MSI with the **O**wned state. Recall that in figure 2.1b, we said

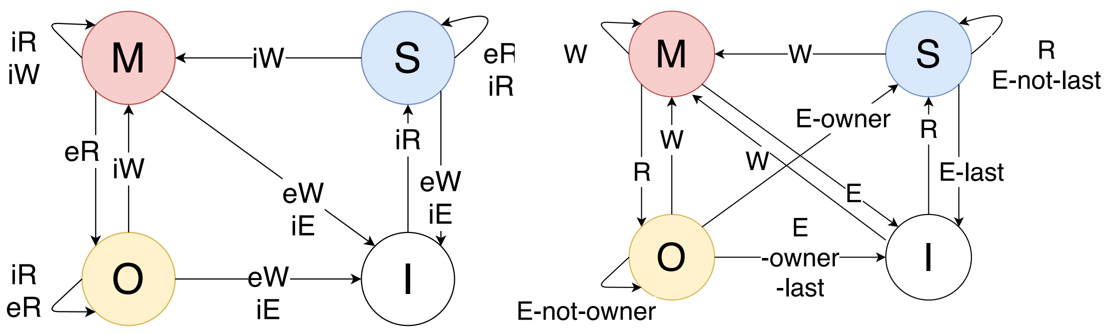
[2]The very keen reader may wonder why we even bother with the E state: why not just make the cache go to M straight away? The reason is that when the cache is in E and evicts the line, it does not need to send the data back to memory; it knows the line is clean. This reduces the bandwidth taken up on the interconnect.



(a) MESI cache controller (b) MESI directory controller

Figure 2.3: MESI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine).

that writing back the value to memory was *optional*. What we really meant is that it is up to the design of the protocol: in MSI and MESI the value would be written back, but in MOSI it would not. Instead, the processor which used to hold the line in Modified state is sent to the Owner state; indicating that whenever another processor wants to join the sharers, it is up to this Owner to forward the data to them. MOSI is as such an optimization of MSI which seeks to reduce the latency of accessing widely shared resources in systems where it is cheaper to send data between the caches than between the directory and the caches. We give the abstract specification of MOSI in figure 2.4.



(a) MOSI cache controller (b) MOSI directory controller

Figure 2.4: MOSI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine).

In each of these protocols, the directory uses two data structures per cache line to keep track of who is doing what to the line. The first is the *sharers vector*: in its simplest form it is a bit-vector with one bit per cache in the system, where the bit is set if and only if that cache holds the line in Shared state. The second is the *owner pointer*, which stores the ID of the cache currently "owning" the line (holding it in `O`, `E`, or `M` state). We will refer to these two data structures as the *auxiliary* state of the directory. To contrast it with the *explicit* state of the directory (whether it is in `M`, `S`, `I`, etc.), we sometimes also refer to it as the *implicit* state. The caches themselves also store auxiliary state in the form of a counter to keep track of how many other caches it is waiting for replies from.

As was alluded to earlier, the specifications of these three protocols which we have given thus far would not quite qualify as SSPs. To make them do so, we would need to specify how the flow of messages and the handling of the auxiliary state implement the communication between the cache controllers and the directory controller. For readers of this dissertation, it is not necessary to know in detail how this is done for MOSI and MESI. However, it is helpful for the reader's understanding of chapters 3 and 4 in particular if they are familiar with the complete stable-state specification of MSI, as given in appendix B. We therefore invite readers who are perhaps not very comfortable with cache coherence protocols to spend some time familiarizing themselves with the contents of this appendix, and also recommend to them that they keep appendix A in mind when reading the later chapters of this dissertation. This appendix gives a quick index of the messages which are used in the protocols considered here, which may serve as a helpful reminder to the reader.

We conclude this section by noting that all three of these protocols would of course also need transient states if they were to handle concurrency. We will not give their complete specifications here, since doing so is a tedious process which would require us to go into way more detail than is necessary for this dissertation; the keen reader may instead find them in Nagarajan et al. (2020). It is for the purposes of this dissertation sufficient that the reader is familiar with the convention that the transient state `XY` refers to when the machine was initially in `X`, requested to go to `Y`, but has not yet heard back from everyone.

2.3 Murphi

Murphi (sometimes stylized as Murϕ) is an enumerative model checker which was first introduced by Dill et al. (1992). In this dissertation, we make heavy use of Murphi to verify our cache coherence protocols, and we will therefore spend some time introducing it in detail.

Murphi has its own input language (somewhat confusingly also called Murphi), in which the user specifies the system, its transitions, and the facts (invariants) to verify it against. In the words of the Murphi Annotated Reference Manual (R. Melton, 1996):

> A Murphi description consists of declarations of constants, types, global variables, and procedures; a collection of transition rules; a description of the initial states; and a set of invariants.

The Murphi language allows for some common datatypes, such as booleans and integers (but only when bounded in a finite range, to make the state space enumerable). In addition, it includes special types such as the `SCALARSET`, which can be used for symmetric (semantically permutable) sets of objects. For example, the set of caches in a system can be permuted, because from the point of view of space exploration, having seen cache A in state M and cache B in state I is the same as having seen cache A in state I and cache B in state M. This symmetry reduction is the key to how Murphi makes the verification procedure tractable; we refer the interested reader to the details given by Dill et al. (1992), along with more recent versions of Murphi such as CMurphi (Della Penna et al., 2002).

Given a specification, the Murphi model checker verifies the system by enumerating all possible states (up to symmetry reduction) and checking them against the specified invariants. If an invariant fails or the system encounters a deadlock Murphi ends the verification procedure and presents the user with the violating trace (that is, sequence of states). This trace is often called a *counter-example* or *falsification* of the desired properties.

Later in chapter 3, we will use some key insights into the structure of Murphi's state space exploration to verify Stable State Protocols without concurrency. We therefore encourage the reader to familiarize themselves with the way in which Murphi explores the state space; we present our understanding thereof in algorithm 1.

Algorithm 1: Murphi verification procedure (Breadth-first)

```
Result: Success, or a counterexample in the form of a trace of states
$S$ := [initial system state] ;                ▷ A trace of system states
$O$ := {initial system state} ;                ▷ Hashes of all states observed
$Q_S$ := [$S$] ;                               ▷ Queue holding traces to be expanded
while $Q_S$ is not empty do
    $S$ := dequeue entry from $Q_S$ ;
    $s$ := the last entry of $S$ ;             ▷ The most recent system state
    for every invariant $I$ do
        if $s$ does not satisfy $I$ then
            return $S$ ;   ▷ Have found a counterexample to an invariant
    $R$ := all rules whose conditions are true in $s$ ;
    if $R$ is empty then
        return $S$ ;                           ▷ The system is deadlocked
    else
        while $R$ is not empty do
            $r$ := pop a random rule from $R$ ;
            $n$ := the state obtained by executing rule $r$ in state $s$ ;
            $h$ := hash of $n$ under symmetry reduction ;
            if $h$ is not in $O$ then
                $N$ := $n$ appended to $S$ ;
                add $h$ to $O$ ;
                enqueue $N$ in $Q_S$ ;
            else
                ▷ $n$, or a state equivalent to it, has already been
                  explored; take no further action in this iteration
return Success
```

2.4 ProtoGen

ProtoGen (Oswald et al., 2018) is the most advanced work towards automating the design of cache coherence protocols currently found in the literature. It is an algorithm (and a piece of software implementing said algorithm) which given a correct stable-state specification of a directory cache coherence protocol generates a fully concurrent version of the protocol. Compared to the approaches we will briefly discuss in 2.5, it does not rely on user intervention during synthesis, and it avoids the state space explosion thereof by using the insight that the directory controller is a linearisation point for the messages in the system. ProtoGen also has a backend which outputs Murphi code for the generated protocol. As such, the verification process is intimately integrated into ProtoGen.

For an example of how ProtoGen works, suppose a cache is in state `I` and issues a `GetS` message to the directory in order to gain read permission. It should now leave state `I` to remember it has started a transition to another state, yet it cannot yet be in state `S` because the directory has not yet responded. We therefore generate a new transient state `IS`, which the cache goes to. Now, another cache decides to go to `M` and sends a `GetM` to the directory. From the specification of the interconnects, ProtoGen can deduce that it is possible that this `GetM` will reach the directory before the directory's `GetS_Ack` has reached the original cache. In this case, the original cache might while in state `IS` receive an `Inv` (invalidation) message from the directory. We therefore generate a new transient state, `ISI`, which signifies that the cache while in `IS` received an `Inv`, is functionally in state `I`, but is still waiting for the `GetS_Ack` from its previously attempted `I`$\rightarrow$`S` transition.

This type of deduction can be extended to obtain all possible races which arise (whether the networks are ordered or unordered), and as long the SSP is correctly specified ProtoGen will use these to obtain a correct concurrent protocol. Unfortunately, ProtoGen does not specify what it means for the SSP to be correct, and it does not do anything to help with its design; it is this which we seek to remedy in this dissertation.

2.5 Other Related Work

The earliest publicly presented work which sought to reduce the complexity of writing cache coherence protocols was probably Teapot (Chandra et al., 1996). Teapot is a Domain-Specific Language (DSL) which was designed to make writing protocols easier. Compared to general programming languages such as C writing a cache coherence protocol in Teapot requires less boilerplate code, but Teapot does not do anything in terms of synthesis. Like ProtoGen, Teapot can output the protocol in a format suitable for a model checking tool.

TRANSIT (Udupa et al., 2013) takes an iterative approach to synthesising cache coherence protocols. First, the user defines a skeleton implementation of the protocol, similar to the SSPs taken as input by ProtoGen. In this skeleton, they may leave several *holes* to be filled in by TRANSIT, although they must provide snippets showing the expected behavior of the holes. TRANSIT then synthesises fixes to the holes, using the snippets; to avoid state space explosion, each hole is synthesised independently. If TRANSIT is unable to synthesise a correct protocol, it presents the user with counter-examples which the user must take into account to improve the skeleton implementation of the protocol. Compared to ProtoGen, TRANSIT thus relies on far more intervention from the user.

The most recent pre-ProtoGen synthesis effort which we are aware of is VerC3 (Elver et al., 2018). VerC3 sought to eliminate the iterative nature of TRANSIT, instead relying solely on properties of the protocol to perform synthesis. Without hints from the user, the synthesis state space explodes; VerC3 tried to cope with this by using a candidate pruning procedure. Despite this, VerC3 is only able to synthesise about a third of the holes which may arise due to race conditions in an MSI protocol.

Chapter 3

Defining and Verifying the Correctness of Stable State Protocols

Discussing bugs in a protocol requires a precise notion of what it means for the protocol to be correct in the first place. In section 2.1 we gave the criteria for correct cache coherence protocols and discussed how to verify these using the Murphi model checker, but these criteria are not a fair way to determine the correctness of a Stable State protocol. Naively plugging an SSP into a model checker such as Murphi will always encounter errors as soon as more than one core is allowed due to the ensuing concurrency. Recall the example from section 2.4 where a cache whilst attempting an `I` to `S` transition receives an invalidation message, causing ProtoGen to derive a new state `ISI`. This state is not part of the SSP, so if it were not for ProtoGen then how could the cache know how to handle this scenario? An error would have to be thrown, or the message would have to be ignored (likely leading to a violation of the SWMR invariant further down the line).

Perhaps a more suitable way to define the correctness of a stable state protocol could be to say that the protocol is correct if, and only if, the concurrent version obtained by running ProtoGen on the SSP satisfies the criteria from section 2.1. However, this would impede our future efforts to pinpoint bugs in the stable state protocols, as the concurrency will complicate the counter-examples provided by Murphi.

In this chapter, we begin discussing the core contributions of this dissertation by describing the approach which our branch of ProtoGen uses to verify Stable State Protocols against the SWMR invariant and deadlocks without complicating the error traces generated by Murphi. We begin by motivating and presenting a refined correctness criterion which is suitable for SSPs, before moving on to discussing how to verify if this criterion holds.

3.1 Defining a Correctness Criterion for Stable State Protocols

Even without concurrency, cache coherence protocols need transient states. For example, the MSI protocol contains a cache state `IS` which indicates that the cache was in state `I`, requested to obtain the line with read permission (i.e. going to state `S`), but has not yet heard back from the directory. Such transient states are however merely an implementation detail of the transitions in the stable state specification; unlike transient states which arise due to possible concurrency, they do not add more transitions to the high-level system.

It is useful to be able to discuss when a full set of transitions between non-concurrent states has finished – for example, a full transition from `I` to `IS` to `S`. We therefore define the notion of a *transaction* in definition 3.1; a similar definition is given in Nagarajan et al. 2020.

> **Definition 3.1.** A **transaction** is the period in (logical) time which begins with any attempted read of, write to, or eviction of a cache block, and lasts until all (if any) coherence events expected to arise due to that action have taken place.

Using the notion of a transaction allow us to concretely specify a sense of logical atomicity in a protocol, which we dub *transaction-atomicity*:

> **Definition 3.2.** A system is **transaction-atomic** if, and only if, at any given time there is at most 1 transaction active per cache block.

As a sanity check: is this definition meaningful (i.e. are systems typically not transaction-atomic)? The answer is *yes*. Because transactions themselves often consist of several messages being sent between different actors, multiple transactions can end up overlapping even if they are targeting the same address. An example of such a situation is given in figure 3.1. Here P0 wants to write to the cache line, and thus begins a transaction. However, before it has acquired read-exclusive (i.e. write) permission, a read miss occurs in P1, unwittingly starting a racing transaction. In the example shown P0's transaction wins the race, and it may write before responding to P1's request.

When a protocol is used in a transaction-atomic system, only some of its transient states will come into play. Consider figure 3.2, which contains all of the transitions which can arise in the MSI protocol when a core tries to write to an address which it currently holds in state `I`. The number of states and transitions involved in the full protocol is large, but only a much smaller amount is relevant when the target system is transaction-atomic. These transitions are precisely those that, like the previously discussed `IS` state, implement one of the transitions specified in the SSP.

This reasoning tells us that running an arbitrary protocol in a transaction-atomic system makes it logically behave just like its stable state representation. Thus, it seems fair to say that the stable state representation is *correct* if and only if the full protocol which it can be mechanically extended to ”works” in transaction-atomic systems. If we take ”works” to mean ”provides coherence as in definition 2.3”, this definition of

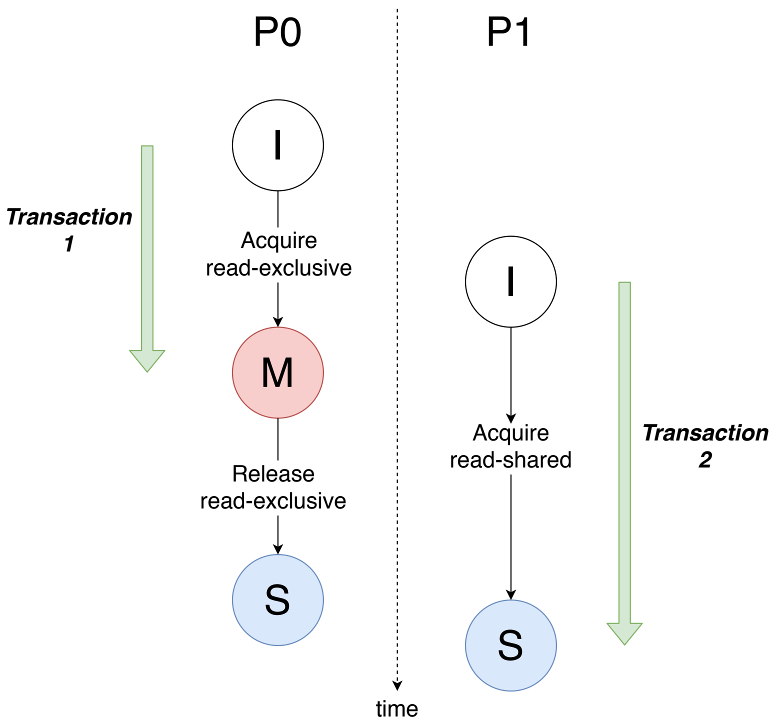


Figure 3.1: Two racing transactions to the same address from two different processors. Note that P0 releasing its read-exclusive permission to the cache line is part of P1's transaction, because it is a coherence event which is occurring as a result of P1 attempting to read the line.

correctness is a clear improvement from where we started off: it allows us to verify an SSP in the Murphi model checker to determine whether it is correct, and yet we do not need to worry about the correctness of the concurrency-induced transitions generated by ProtoGen. However, this criterion is not complete; we can do better.

First of all, we argue that SSPs should be assumed to never be dependant on *stalling*. Stalling is when a machine receives a message and finds it is not yet ready to deal with it, putting it back on the network to be addressed at a later time. Stalling is a neat trick from the point of view of simplifying protocol design, but hinders performance, and is therefore to be avoided if possible; Nagarajan et al. (2020) discuss the impacts of stalling on performance more thoroughly. Whether or not going through the extra effort of producing a non-stalling protocol is worth it when designing a full protocol from the ground up, there should be no reason to rely on stalling in SSPs, which are already small and relatively simplistic protocols. Instead, stalling within the transitions of an SSP shows that the designer has not adequately considered all possible sequences of events, and an error should therefore be thrown to alert them of this.

Secondly, whether a protocol is an SSP or not it should have the property that for every machine in the system, there is a sequence of events which takes that machine from its initial state into any of its other states. If this is not the case and there thus exists some state which will never be reached in the system, the protocol is poorly specified in the sense that it contains superfluous states (in the best case wasting precious low-latency

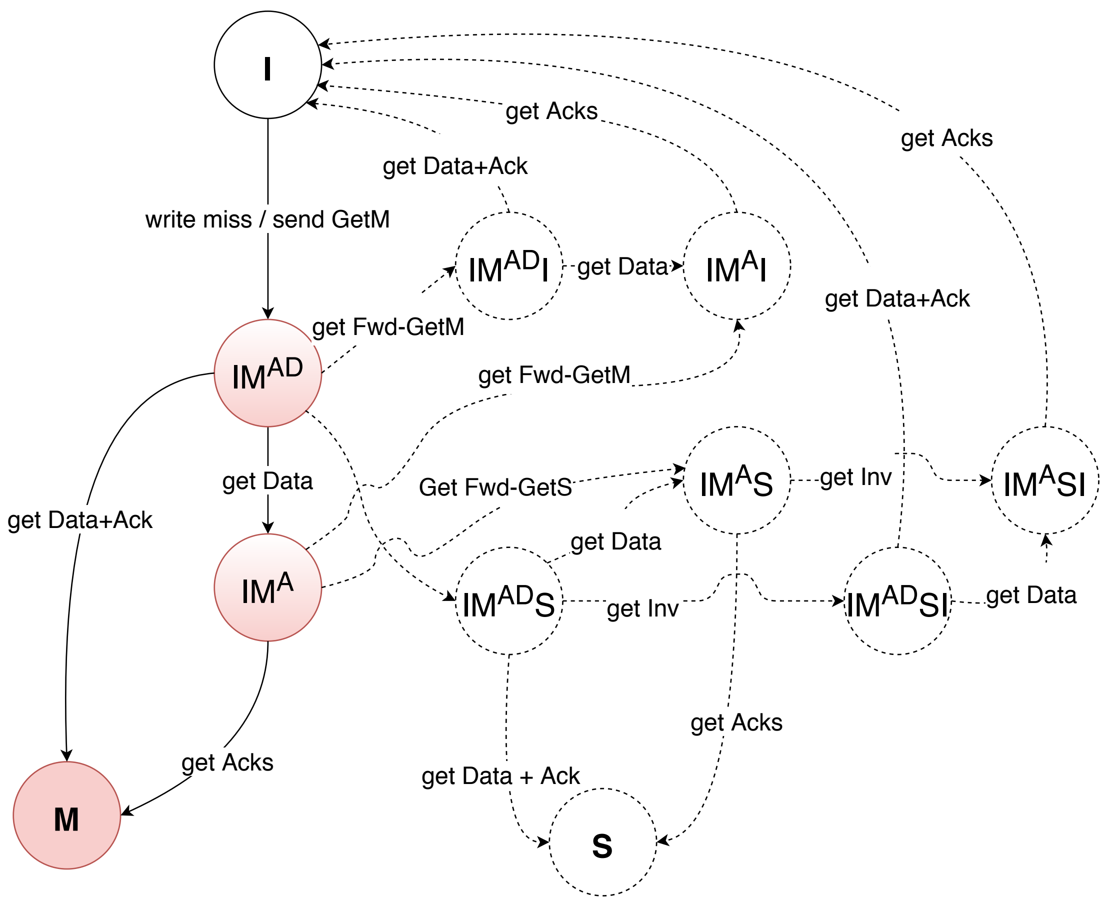


Figure 3.2: MSI protocol transitions arising from a write miss in I state. Transitions which occur in transaction-atomic systems in solid, those that do not in dotted lines. Based on material from [Arm Research] (2018).

memory, and in the worst case being a symptom of much more serious issues). For example, say that the designer has specified that a protocol contains a directory with possible states `I`, `S`, `M` and `E`, and that the directory is initially in state `I`. There must then be some sequence of transitions which causes the directory to go from `I` to any of `S`, `M` and `E`.

Taking both of these additional criteria into account in conjunction with what we had before, we have finally arrived at a definition of correctness which accurately and completely models what we expect from a stable state protocol:

Definition 3.3. A Stable State Protocol (SSP) is said to be **correct** (or bug-free) if, and only if all three of the following criteria hold:

- For every machine in the system, all of its states are reachable from its initial state;
- The protocol obtained by running ProtoGen on the SSP is found to provide coherence (definition 2.3) for transaction-atomic systems;
- The protocol does so without ever stalling.

3.2 Verification of the Criterion

In our branch of ProtoGen, the first step taken to verify the input SSP against the criteria in definition 3.3 is to check the state reachability criterion. This is done by exploring the system in a depth-first fashion. Pseudo-code for the approach taken is given in algorithm 2. This algorithm recursively builds a set of states which are reachable from the initial state of each machine, and then ensures that every stable state of that machine has been visited.

Algorithm 2: State reachability pseudo-code.

```
for every machine type $M$ do
    $I$ := the initial state of $M$ ;
    $T$ := a map from states to sets of states s.t. $S_2$ is in $T[S_1]$ iff $M$ has a transition
      from $S_1$ to $S_2$ ;
    $V := \{\}$ ;    ▷ Set of states visited in the exploration process,
      initially empty
    exploreFromVertex($T, I, V$) ;    ▷ Recursively explore the system,
      starting at the initial state
    for every stable state $S$ of $M$ do
        if $S \notin V$ then
            return False ;
return True ;
Function exploreFromVertex($S, S, V$):
    if $S \notin V$ then
        add $S$ to $V$ ;
        for every state $S'$ in $T[S]$ do
            explorefromVertex($T, S', V$)
```

Having ensured that the first criterion holds, we move on to verifying the second and third criteria. This is done using the Murphi model checker and checking for SWMR (definition 2.1) breaches and deadlocks, much like how the full protocol is verified once ProtoGen has finished. The difference is that the Murphi system model is made transaction-atomic and that the protocol is specified in such a way that if it ever needed to stall a message, an error would instead be thrown.

To make the system model transaction-atomic, each transaction is treated as a critical section. When a cache wants to read from, write to, or evict a cache block, it must acquire a lock for that address. The cache will only proceed with the access if the lock is acquired; if it fails it backs off, and may try again later. Once acquired, the lock will then be held until the transaction is finished.

To see how each transaction is made into a critical section, first recall the structure of Murphi programs as given in section 2.3. Most importantly, recall that the user specifies a number of *transition rules*, consisting of a precondition and an effect. These rules are what Murphi uses to navigate the state space: it picks a rule whose preconditions are satisfied in the current state, executes its effect, and then picks another rule; continuing this process as long as it finds new states to explore. When verifying a cache coherence protocol, rules typically come in two forms: those representing cache accesses (adding messages such as `GetS` or `GetM` to the networks if there is a miss), and those that pop messages from a network and invoke the corresponding message-handling routine (often adding messages such as acknowledgements or invalidations to the networks). Using this insight, we can make the system model transaction-atomic through two simple changes:

1. All cache access rules have their preconditions extended with a check ensuring that the lock for that address is currently free, and their effects are extended with an acquisition of the lock.

2. A new type of rules, *unlock rules*, are introduced into the system. The only effect of these is that the lock for the address is released. More interestingly, their preconditions require that all of the following holds:

   (a) The cache firing the rule is currently the owner of the lock;

   (b) The cache firing the rule is currently in a stable state;

   (c) Every directory in the system is currently in a stable state;[1]

   (d) Every network in the system is currently empty.

It is worth noting that acquisitions of locks are implemented as a sequence of a read (in the precondition of the rule) and a write (in the effect of the rule). This is safe because Murphi already guarantees that rules are executed atomically; no atomic read-modify-write instruction is necessary.

It is easy to see that the above modifications guarantee transaction-atomicity: the address-specific lock is acquired upon any cache access, and released only when the system has processed every subsequent coherence event arising due to that access. During the time that the lock is held, no cache access rules to the same address can be executed, because their preconditions will not hold. Thus, at most one transaction can be active, per address, at any time.

Having made the above changes, running Murphi will verify criteria 2 and 3 from definition 3.3 for us. Thus, if it does not find any errors, we have verified that the SSP is correct and can move on to verifying the full protocol.

[1]Requiring *every* directory to be in a stable state allows us to ignore scenarios where we might have multiple directories, each being responsible for some set of cache lines.

Chapter 4

Classifying Bugs in Stable State Protocols

Having arrived at a concrete notion of what it means for a Stable State Protocol to be buggy, we can now turn our attention to *how* it may be so. When we make mistakes in our specifications of SSPs, what do those mistakes look like? Can we identify certain *classes* or *groups* of bugs?

In this chapter, we set out to answer these questions. Since there is to our knowledge no existing data to base our arguments on, we begin by presenting a suite of bugs, or micro-benchmark for bug localization, created manually by us. Using insights from this bug suite, we then derive a taxonomy of the types of bugs which we have observed.

4.1 Bug Suite / Micro-Benchmark

The bug suite is an effort to construct an initial dataset from which heuristics for SSP bug localization can be derived, and on which they can be evaluated. In other words, it is an attempt to answer the first of our questions: *when we make mistakes in our specifications of stable state cache coherence protocols, what do those mistakes look like?* In section 4.3, we will briefly discuss how good of an answer this bug suite really is.

The suite consists of 31 protocols, all of which are variations on the sample MSI protocol provided in the ProtoGen source code (ICSA-Caps, 2018). Tables 4.1 and 4.2 list each protocol/test by which machine makes the mistake and what the last cache event or message received was, and also give a short description of each bug.

While we believe most of the tests to be self-explanatory, some require additional explanation. Consider for example tests 18 and 22. Both involve caches in `M` not sending a `WB` (writeback) message to the directory upon receving a `Fwd_GetS`; that is, when the system transitions from having a single read-exclusive cache to several read-shared caches. The difference is that in test 22, the directory still *expects* the cache to send the writeback. While this should quite obviously cause a deadlock under a transaction-atomic (definition 3.2) system (as the directory will never enter a stable state again, and hence the transaction cannot finish), one might wonder why bug 18 would cause any issues at all. After all, we stated in section 2.1 that we are not concerned with the propagation of values through the memory epochs, but rather only with maintaining the SWMR invariant and avoiding deadlocks. It turns out that the writeback is a crucial component in the transition because it lets the directory know that the previous owner has agreed to give up its read-exclusive permission; without this, issues arise when concurrency is added to the system.

Another test which warrants further explanation is number 17. Reading in a read-exclusive state is perfectly fine, so why should we consider this to be a ”bug”? The reason is that if caches always request read-exclusive permission to read lines, read-shared permission will never be granted. In other words, if caches always request to go to `M` when they face a load in `I`, the state `S` will never be reached! This violates the very first criterion of our definition of correctness for SSPs (definition 3.3).

Having now answered the first of our questions, let’s move on to the second: *can we identify certain classes or groups of bugs?*

4.2 Taxonomy

Inspecting the bug suite, some patterns become emergent. For example, many bugs come down to not updating, or erroneously updating, some aspect of the system state (whether the explicit state of a machine or some implicit aspect of the state such as the sharers vector). Meanwhile, others come down to sending the wrong type of messages, or sending messages to the wrong machine. Finally, a small minority do not appear to belong to either of these two groups. Instead, they are bugs which correspond to fundamental problems in the design of the protocol, or which only cause very subtle errors when there is concurrency present (like the previously mentioned test 18).

Using these insights we propose a taxonomy consisting of four classes of bugs, given in figure 4.1. Note that this taxonomy only applies to those bugs which only cause issues at run-time; that is, it does not consider static bugs. We will devote the remainder of this section to looking at each of these four classes in detail, deferring further discussion regarding the validity of the taxonomy until section 4.3.

Taxonomy of bugs in cache coherence protocols
Cache coherence protocols can contain the following types of bugs:

1. Message Bugs
2. State Bugs
3. High-Level Design Bugs
4. Concurrency Bugs

Figure 4.1: Taxonomy of bugs in cache coherence protocols.

Test number	Machine	State	Received Message or Event	Description
1	Directory	I	GetS	Do not add cache to the sharers vector
2	Directory	I	GetS	Do not update state to S
3	Directory	I	GetS	Combination of previous two
4	Directory	I	GetM	Do not update state to M
5	Cache	IS	GetS_Ack	Do not update state to S
6	Directory	M	GetM	Do not add old owner and new sharer to sharers vector
7	Directory	M	PutM	Do not send Put_Ack back
8	Directory	M	PutM	Do not update state to I
9	Directory	M	GetM	Do not update owner pointer
10	Cache	IM	GetM_Ack	Go to M without waiting for Inv_Acks
11	Cache	S	Eviction	Do not send out any messages, nor wait for any; just go to I
12	Cache	M	Fwd_GetS	Go to I instead of S
13	Cache	M	Eviction	Go to S instead of I
14	Directory	S	GetM	Do not clear sharers vector
15	Cache	M	Fwd_GetS	Send unexpected GetS_Ack to directory
16	Cache	M	Fwd_GetM	Do not update state to I
17	Cache	I	Load	Request read-exclusive transition (to M) instead of read-shared (to S)
18	Both	M	External load	Writebacks are no longer performed in system M to S transition

Table 4.1: The manually created bug suite (part 1).

Test number	Machine	State	Received Message or Event	Description
19	Cache	S	Store	Allowed store to occur in state S
20	Cache	IM & SM	Store	Removed declaring that Inv_Acks may arrive before GetM_Ack
21	Cache	S	Inv	Do not send Inv_Ack
22	Cache	M	Fwd_GetS	Do not send WB to directory
23	Cache	S	Eviction	Do not wait for Put_Ack before going to I
24	Directory	S	GetS	Do not add new sharer to sharers vector
25	Cache	IM	GetM_Ack	Do not model the fact that all Inv_Acks might already have arrived
26	Cache	I	Inv_Ack	Forget to increment acknowledgement counter
27	Directory	S	PutS	Go to I even if there are still other sharers
28	Directory	M	PutM	Go to I without checking that the source of the PutM is the current owner
29	Cache	M	GetM	Go to S instead of I
30	Cache	M	Load	Go to S (without any messages sent)
31	Directory	S	GetM	Do not check if the requester was already a sharer when setting the number of invalidations it should expect

Table 4.2: The manually created bug suite (part 2).

4.2.1 Message Bugs

Message bugs are those which relate to the specification of which messages to send at what point in the protocol. This includes sending unexpected messages, sending messages to the wrong machine, and not sending a message when expected to. For example, failing to send out an `Inv_Ack` upon receiving an `Inv` message would be a message bug. Sending it to the directory instead of the cache wanting to transition to `M` state would be another.

We find that message bugs can manifest themselves at run-time through deadlocks or, when we disallow stalling of incoming messages, to ”unexpected message” errors. It is also possible, although less likely, that a message bug will lead to violating safety (SWMR); this might for example happen if the directory upon receiving a `GetM` request in state M sends a `GetM_Ack` back to the requester instead of sending a `Fwd_GetM` to the current owner of the line, leading to two caches having read-exclusive permission simultaneously. In our testing, message bugs typically become obvious within the same transaction as they occur in, when checked in a transaction-atomic system. That is, the bug itself usually lies in the transaction in which the error was thrown. In chapter 5 we will use this insight to find message bugs at the granularity of a single transaction.

In the bug suite, there are 4 tests which we identify as message bugs: 7, 15, 21, and 22.

4.2.2 State Bugs

State bugs are those which relate to the setting and updating of state in the protocol, whether explicit state (such as `I`, `S`, or `M`) or implicit state (such as the owner pointer, the number of acknowledgements which a cache is currently waiting on, etc.). For example, if a directory forgets to update the owner pointer when receiving a `GetM` in `M`, this would be a state bug. Another example would be if a cache upon receiving an `Inv_Ack` forgets to increment its local counter which keeps track of how many such acknowledgements it has received.

State bugs manifest themselves at run-time through either deadlocks, unexpected messages, or safety violations. Unlike message bugs, state bugs often lead to errors much later than the transaction in which the actual error occurred. For example, suppose a directory in `I` receives a `GetS`, but stays in `I` instead of updating its state to `S`. The transaction still finishes without error. However, if the next transaction is a `GetM` from another cache, then the directory will erroneously think there are no sharers of the line and give the new cache read-exclusive permission, violating SWMR in the process.

In the bug suite, there are 19 tests which we identify as state bugs. These are: 1, 2, 3, 4, 5, 6, 8, 9, 12, 13, 14, 16, 24, 26, 27, and 29.

4.2.3 High-Level Design Bugs

High-level design bugs are those bugs which, as the name suggests, really boil down to a lack of fully understanding the target system and how to design a protocol for it. As such they are more serious bugs, which are unlikely to be caused by for example typos. Example High-Level Design Bugs include those which mess up the permissions of the states (e.g. allowing stores in `S` state), not fully declaring every possible order in which a group of messages may arrive at a machine (for example not declaring that `Inv_Acks` may arrive before the `GetM_Ack_AD`), and performing silent transitions when the system is not set up to handle this.

At run-time, high-level design bugs typically either break safety or cause unexpected messages to be received. They can also cause static errors, such as breaking the state reachability requirement of the SSP correctness criterion (definition 3.3).

In the bug suite, there are 8 tests which we identify as High-Level Design Bugs: 10, 11, 19, 20, 23, 25, 30, and 31.

4.2.4 Concurrency Bugs

Finally, we have concurrency bugs. These are the bugs which only cause errors when the system is **not** transaction-atomic. One interpretation is thus that they are the complement of the union of the above three classes of bugs (all of which cause issues in transaction-atomic systems).

Concurrency bugs are interesting because they show that there is a hole in our reasoning thus far. We have assumed that given a correct SSP, ProtoGen generates a correct concurrent protocol. We have also derived a definition of what it means for an SSP to be correct (definition 3.3), which we have argued is well-formed and covers everything we expect from a stable state protocol. However, protocols which contain (only) concurrency bugs do *not* breach the definition of correctness in SSPs, and yet running ProtoGen on them will *not* lead to correct concurrent protocols! This is a symptom of a fundamental limitation of ProtoGen as a synthesis tool: while powerful, it cannot automatically derive *everything* necessary for a protocol to handle concurrency. Much like the designer still has to declare the order in which messages may arrive at a machine, they also still need to design sequences of messages which are capable of handling the concurrency in the system. ProtoGen can only fill in the holes in the design which arise due to races on the interconnects.

Concurrency bugs are a diverse type of bug, and as such they can lead to any type of issue in the full protocol.

In the bug suite, there are 2 bugs which we identify as concurrency bugs: 18 and 28.

4.3 Discussion: Shortcomings of the Bug Suite and the Taxonomy

We have now introduced the bug suite, a first attempt at constructing a dataset of buggy protocols. We have also argued for a taxonomy of these bugs, and have discussed in detail what each of the classes look like. Now, we will deal with the elephant in the room: the shortcomings of the work presented in this chapter.

First of all, the proposed taxonomy is not a complete characterisation of every type of bug possible in cache coherence protocols. Therefore, it may not always be straight-forward to assign a class to a given bug. In our eyes, the Venn diagram in figure 4.2 captures the relationship between the classes. This shows that especially the ”high-level design” class is somewhat poorly defined. For example, at what point does a bug become so severe that it should be classed as a high-level design bug? If you have a bunch of state bugs, how many does it take before they jointly become a high-level design bug? If the designer completely messes up the entire flow of messages in the system, is that a large set of message bugs or a single high-level design bug?

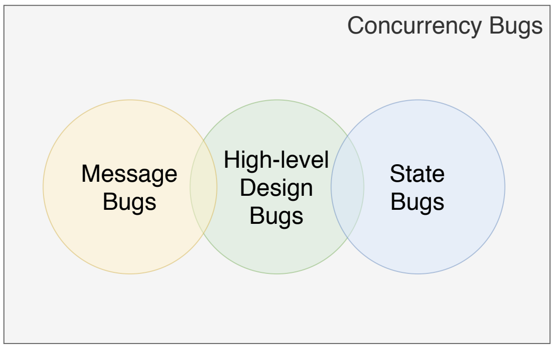


Figure 4.2: Venn diagram of the taxonomy in figure 4.1.

Secondly, it may seem that the different classes in this taxonomy are underdeveloped. By this we mean that further investigation may reveal that these are in fact made up of nuanced and distinct sub-classes. This holds especially true for high-level design bugs and concurrency bugs, which are defined by their symptoms rather than their causes. We conjecture that such further analysis could very well reveal useful insights, but it would require a dataset far broader than that considered here.

Finally, we recognize that at the end of the day, a taxonomy of bugs in cache coherence protocols would be a lot more convincing if it was based on a dataset collected in the wild, or at least not created by the same authors as the dataset itself.

Despite these issues, we find the taxonomy presented in this chapter to be a useful tool when discussing cache coherence protocols, and it is one which allows us to derive some interesting bug localization methods (which we will discuss in the next chapter). One common thread in this project is the fact that we are carving out a new direction of research in this field; as such, we rarely have the luxury of relying on existing literature or established practices. We hope that even if future work proves that the work presented in this chapter is incomplete or even misleading, this setting stone which we have laid down can be of aid to future research in this field.

Chapter 5

Methods for Debugging Stable State Protocols

Having established a concrete sense of what it means for a stable state protocol to be incorrect (in chapter 3), and having also constructed a suite of incorrect protocols along with a taxonomy thereof (in chapter 4), we now turn our head towards what can be done *when* a stable state protocol is incorrect.

Falsification engines such as Murphi can give error traces with concrete examples of how the protocol has failed to satisfy some desired property. However, parsing the error traces (to actually understand where the bug lies) is a laborious task, and one which often involves shifting through completely irrelevant transitions to find the root cause of the error.

In this chapter, we introduce two alternative methods for finding bugs in SSPs. The first one, *Reduced Model Checking*, is a twist on model checking which enables us to home in on message bugs without considering traces of more than 1 transaction at a time. The second, *State Consistency Tracking*[1], exploits one key observation about the protocols to find state bugs before they cause violations of safety. Both of these methods are heuristics; as we will see, they cannot catch all bugs nor work under all conditions. However, we believe these heuristics can still prove to be powerful debugging aids, and we devote significant portions of this chapter and the next to justifying these beliefs.

5.1 Catching Message Bugs with Reduced Model Checking

Recall from section 4.2.1 that a message bug is one which relates to the specification of which messages to send at what point in the protocol. Recall also that in the same section, we stated that in our testing message bugs typically manifest themselves as errors within the same transaction as they occur: so surely debugging them should

[1]Disclaimer: State Consistency Tracking (SCT) as a method for finding bugs was originally envisioned in collaboration with Nicolai Oswald of ProtoGen (Oswald et al., 2018). The concrete implementation of SCT in our branch of ProtoGen was wholly constructed by us.

be easy, right? The problem is that you can certainly focus your attention on the last transaction in the error trace, but only *if* you already know that the bug is of the message type. When debugging a protocol in the wild, you will not have the luxury of knowing this a priori. However, if we think about the definition of message bugs a bit more, we can make a rather revealing observation:

> **Observation 5.1.** Message bugs are completely separate from the update of auxiliary state in the protocol; therefore, they will occur whether this state is accurately tracked or not.
>
> **Remark.** If this observation is not immediately clear to the reader, consider the implications of having a message bug which *did* depend on auxiliary state; for example, if not updating the sharers vector upon evictions caused invalidation messages to be sent to the (already invalidated) ex-sharers. Then the real cause of the bug would not be the specification of the message itself; instead, it would be the erroneous update of auxiliary state. Thus, the bug would per our taxonomy be classed as a state bug, not a message bug.

This observation is the key motivation for a method which we call *Reduced Model Checking* (RMC). In abstract terms, RMC simply carries out model checking without tracking the auxiliary state in the system, instead inferring from the user's specification what behavior is possible *if* the auxiliary state is handled correctly. Since RMC does not track auxiliary state, it therefore does not need to consider the history of past events at all; it only has to work with one transaction at a time, thus by construction singling out message bugs at the granularity of a single transaction.

Put another way, intuitively RMC aims to test whether the specifications of message flows in the cache controllers and the directory controller are consistent *assuming everything else goes right*. If they are not, there is most certainly a bug present: if the very specifications of message flows do not match when the auxiliary state is handled correctly, how could it possibly match at run-time? At best, the system will at run-time be handling auxiliary state perfectly (as expected by RMC); at worst, it introduces more issues by not not updating the auxiliary state correctly.

Concretely, RMC works by implementing the following steps:

1. Turning the deterministic e-FSMs (extended finite state machines) associated with each transaction into non-deterministic FSMs (NFAs), keeping the same transitions but dropping the conditions based on auxiliary state and making every stable state accepting

2. Considering each possible combination of such NFAs under the designer's specification of the protocol, assuming auxiliary state is handled correctly

3. Simulating each combination of NFAs according to every possible ordering of the messages being sent between them.

If at any point a NFA is found unable to receive a message (for example because the message is being sent to the wrong machine), an error is thrown which alerts the user of what transaction was being investigated, what the ordering of messages was, and which machine was unable to receive the message. Similarly, if RMC makes it to the end of the list of messages being sent but one of the NFAs is not accepting (for example because there is a missing message in the transaction), an error is thrown alerting the user of this. We can also check that whenever a reply is sent to the source of a previous message, that previous message is in scope (e.g. we're not trying to reply to "GetM.src" after receiving a GetS), and throw an error if not. Finally, we can perform the sanity check that machines never send multiple messages to the same target within a single transition (which indicates there is a surplus message present, since otherwise the two messages would simply be joined to one).

Algorithm 3 reiterates this high-level overview of RMC in concrete, algorithmic terms. This representation leaves out some details, such as how the two functions `makeNFATree` and `toLeafList` derive every possible combination of NFAs in the specification of the protocol. We find it most illustrative to explain these details through an example, rather than overwhelming algorithmic detail. As such, we will now walk through a concrete application of RMC: verifying a cache's `I` to `M` transition.

Algorithm 3: Reduced Model Checking (high-level overview)

```
Result: Success, or an error
for every stable cache state s and every action a in s do
    nfa_tree = makeNFATree(s, a) ;
    nfa_combos = toLeafList(nfa_tree) ;
    ; ▷ The above will throw a SrcNotInScopeError if a machine at
      any point tries to send a reply to the source of a message
      which is not in scope, and a MultipleMessagesError if any
      machine within a single transition tries to send more than
      one message to the same target.
    for nfa_combo in nfa_combos do
        for every linear extension M of nfa_combo.partially_ordered_messages do
            for $m \in M$ do
                deliver m to its target in nfa_combo ;
                if the target can never receive such messages as m then
                    ; ▷ For example, an Inv message is sent to the
                      directory
                    return UnexpectedMessageError
                if the target could not receive m in its current state then
                    ; ▷ For example, a cache in I receives an Inv
                    return UnexpectedMessageInStateError ;
            for $nfa \in nfa\_combo$ do
                if nfa is not accepting then
                    return NFANotAcceptingError ;
return Success
```

5.1.1 Example: Verifying a Cache's I to M Transition with Reduced Model Checking

First, RMC needs to find out what transactions may come about as part of a cache's `I` to `M` transition. For example, one possible transaction is the one where the directory upon receiving the cache's request is itself in state `I`, and the transaction thus only consists of the directory receiving the request and sending back an acknowledgement. Another possible transaction is the one where the directory is already in `M`, meaning there is another cache which currently owns the line; this transaction would consist of the directory forwarding the request to the current owner, which in turn would send on the data and acknowledgement to the new owner (and demoting itself to `I`).

To automatically derive every possible such transaction, RMC parses the messages being sent by the machines and creates a tree of sets of NFAs (the `makeNFATree` function in algorithm 3). To do this, RMC starts by looking at the messages being sent in the cache's own transition. Whenever a message is sent to a machine which has not been seen before, an NFA for the new machine is constructed for each stable state in which that machine may receive the message. The tree then branches so that every combination of these new NFAs is added as a new child, and the new NFAs are then in turned inspected for which messages they send. When all the messages have been parsed and no new machines are being encountered, the leafs of the tree contain the full set of NFAs corresponding to the different transactions; `toLeafList` simply extracts these leafs from the tree.

In case the above process is not immediately clear, the tree of sets of NFAs which is derived for the desired `I` to `M` transition is given in figure 5.1. This shows that there are three possible transactions arising from a cache's desired `I` to `M` transition: the two which we discussed above, and one where the directory receives the request while in `S`. This last one showcases an important detail of RMC: when a message is broadcast to a set of machines which varies in size, such as the sharers vector, RMC has no way of telling how many members of that set to simulate (due to not tracking auxiliary state). Instead, RMC adds precisely one *prototype* member of that set. We will discuss the implications of this in section 5.1.2.

Having found the sets of NFAs which make up every possible transaction, RMC then moves on to the task of simulating these transactions. In this example we focus on the case in which the directory receives the GetM in state `S` but, to reiterate, this process is in fact carried out for every transaction.

To simulate the transaction, RMC once again looks at the messages being sent by the machines, this time creating a partial ordering $\leq$ over the set M of messages. For any two messages $m,n \in M$, a partial ordering constraint $m \leq n$ is added if and only if:

- It is receiving m in a machine which causes n to be sent; or,
- *All* of the following four constraints hold:
    1. m and n are two messages sent by the same machine in the same transition, i.e. they are both sent upon receiving some message

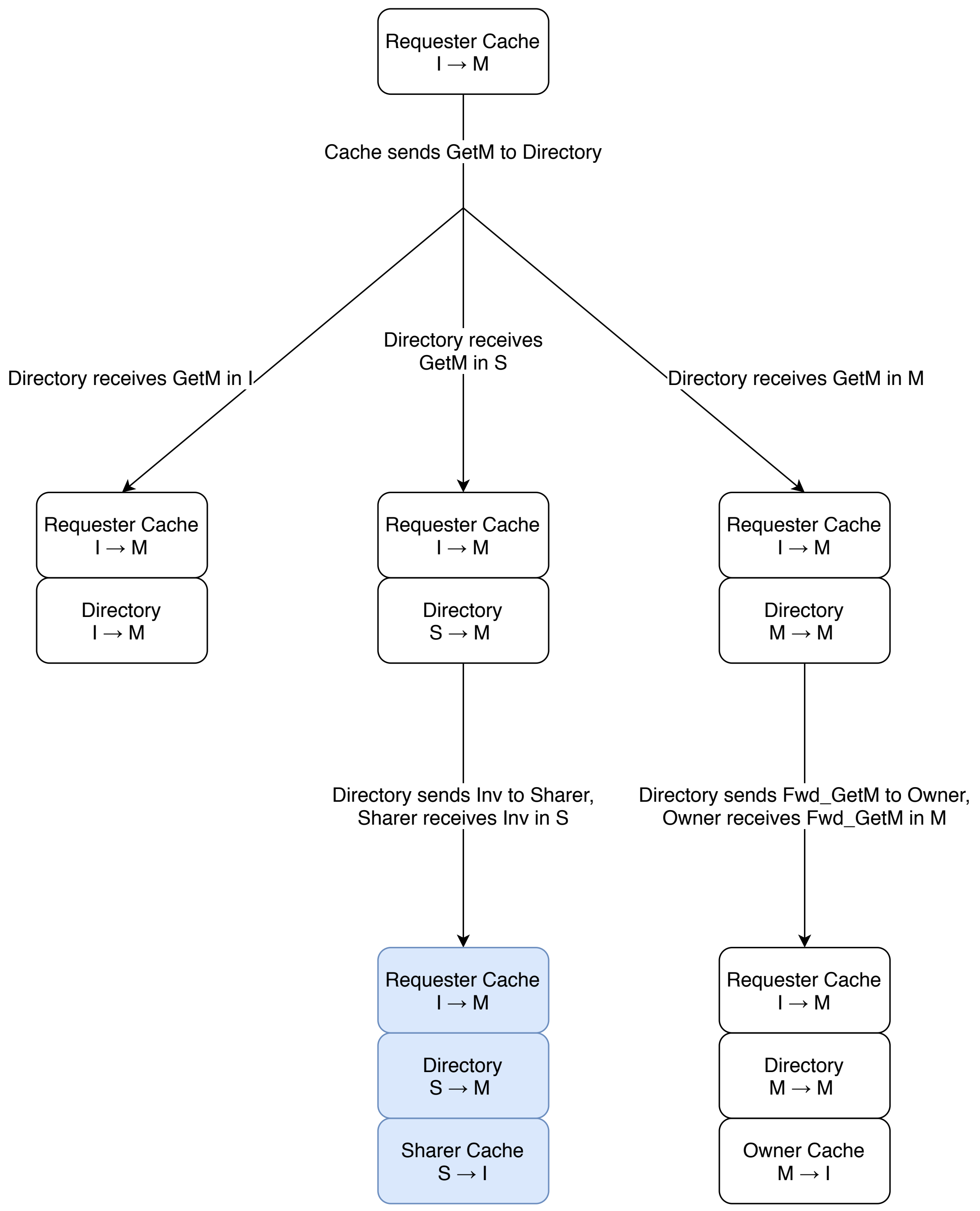


Figure 5.1: The tree of NFAs derived for the I to M transition. The leafs are the sets of NFAs implementing each possible transaction arising from the transition; the leaf in blue is the transaction chosen for the rest of the example.

2. m and n are sent on the same network
3. n is put onto the network after m
4. The network is ordered.

The partial orderings tell RMC in which order the messages may arrive at their targets, so that it knows not to simulate orders which won't come up in practice. In our running example, the partial orderings found are:

$$\{\text{store} \leq \text{GetM}, \text{GetM} \leq \text{Inv}, \text{GetM} \leq \text{GetM\_Ack\_AD}, \text{Inv} \leq \text{Inv\_Ack}\}$$

Having derived the partial orderings over the messages, the next step is to find every linear extension $<$ of $\leq$; that is, every relation $<$ such that for every $m,n \in M$, $m \leq n \rightarrow m < n$. Brightwell and Winkler (1991) show that the problem of finding every such linear extension is #P-complete, meaning no polynomial-time algorithm for this task is known to exist. Luckily, as we saw above, the sets of messages and partial orderings being sent in a single transaction are in practice *very* small. We therefore settle for the factorial-complexity method of simply generating every possible ordering of messages in M and then discarding those which do no satisfy $\leq$. In our running example, we end up with 3 linear extensions of the partial ordering:

1. store $<$ GetM $<$ GetM_Ack_AD $<$ Inv $<$ Inv_Ack
2. store $<$ GetM $<$ Inv $<$ GetM_Ack_AD $<$ Inv_Ack
3. store $<$ GetM $<$ Inv $<$ Inv_Ack $<$ GetM_Ack_AD

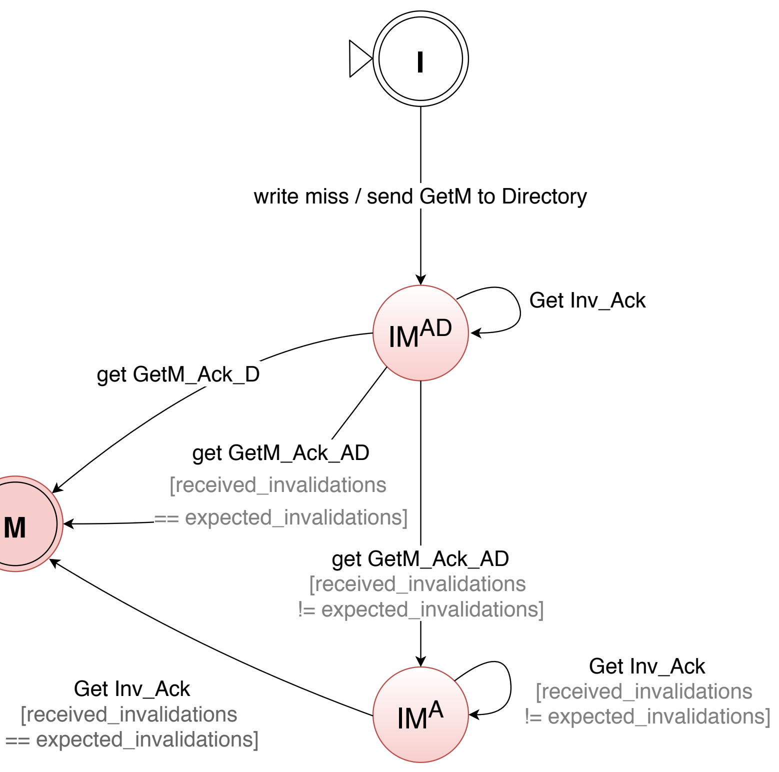
Just like RMC simulates every possible transaction, it also simulates each transaction for every possible ordering of the messages.

The time has now finally come to simulate a run of the transaction, using the desired order of messages. To do this, we take the NFAs for the transaction (figure 5.2) and simply invoke the transitions by delivering the messages in order. In this case, since the protocol is free of bugs, we find that everything goes smoothly; no errors are thrown, and all the machines end up in accepting states.

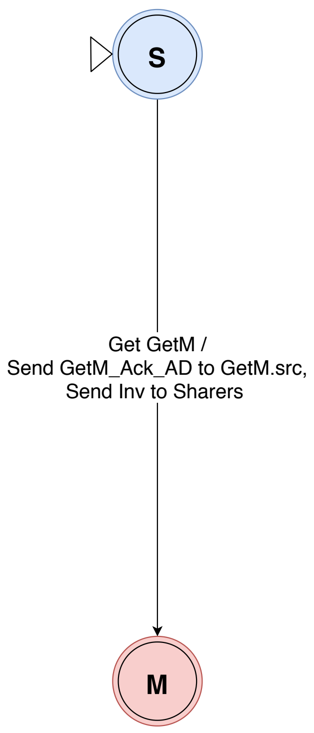
We conclude this example by quickly considering some examples of what would have happened if there *were* message bugs present in the protocol.

First, what if instead of sending a `GetM` to the directory, the cache initially sent a `GetS` instead? Depending on what state the directory was in, either it or a cache currently owning the line would end up sending a `GetS_Ack` to the requesting cache. When simulating the transaction we would thus find ourselves trying to deliver this `GetS_Ack` message at the cache, but there would be no transitions available that receive such a message. Thus, we would throw an error, and alert the user of this problem.

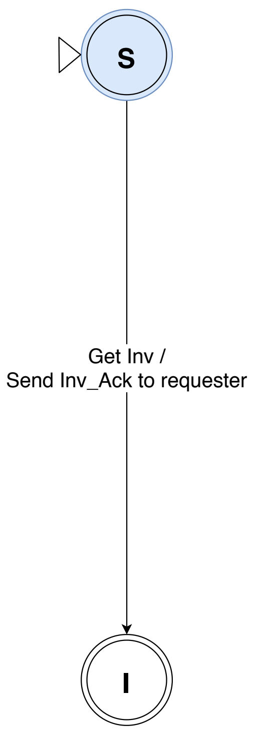
Next, what if the directory sent the invalidation to the requesting cache rather than the sharers? Then we would end up trying to invoke a "Get `Inv`" transition in some point of the requesting cache's NFA (figure 5.2a). Since no such transition exists, we would throw an error in this case too.



(a) The NFA for the cache going I to M. Note that the presence of the A in GetM_Ack_$\{$A$\}$D tells the cache whether it needs to wait for Inv_Acks or not (and if so, how many).



(b) The NFA for the directory.



(c) The NFA for the (prototype) sharer.

Figure 5.2: NFAs for the example transaction in section 5.1.1. Conditions previously present in the e-FSMs but dropped in the NFAs in grey.

Having walked through a detailed example of Reduced Model Checking, we hope that the reader finds the principles behind the method clear, and that they can see how RMC can be used to find bugs in protocols. However, one single example can of course not give a fair picture of all of RMC's strengths and weaknesses, when it shines and when it crumbles into uselessness. In the next section, we therefore finish off our discussion of RMC by discussing its specific strengths and weaknesses from a design point of view, deferring quantitative evaluation until chapter 6.

5.1.2 Strengths and Weaknesses of Reduced Model Checking

The greatest strength of Reduced Model Checking is that it only considers transactions in isolation; it constructs no history (or trace) of transactions, so when a bug is found it is always at the granularity of a single transaction. As we will see in chapter 6, this greatly reduces the average number of transactions which the user has to consider in the case that the underlying bug is a message bug, effectively reducing the amount of irrelevant information the debugger has to shift through.

Another strength of RMC is that although it was designed to detect message bugs, it can also find a small number of state bugs: those where the final state update is not performed in some machine, causing the machine's NFA to not be accepting at the end of the transaction. Of course such bugs only make up a small fraction of the possible state bugs in a protocol, but when they occur RMC is there to help the debugger. Since RMC verifies the transactions for each possible linear extension of the partial order over messages, it can also detect high-level design bugs in which the user has not considered every possible order that messages may arrive at their targets; helping the user write efficient protocols free from stalling.

No heuristic approach to finding bugs is without its shortcomings, and RMC is no exception to this rule. One weakness of RMC is that by ignoring auxiliary state, even some message bugs may be missed. Consider the transaction given as an example in the preceding section, but imagine that the sharer forgets to send the invalidation acknowledgement (`Inv_Ack`) upon being invalidated. When taking auxiliary state into account, this would (in a transaction-atomic system) cause a deadlock where the requesting cache waits indefinitely for an `Inv_Ack` which never arrives. However, when simulating the NFA for the requesting cache (figure 5.2a), receiving the `Inv_Ack(s)` is not necessary to reach the accepting `M` state. Thus, RMC will not notice this bug. At first glance, this may seem to contradict observation 5.1, but in our eyes it does not; the bug still occurs regardless of auxiliary state, it is simply that in turning the deterministic finite state machines into NFAs RMC has lost information about which paths will be taken in the system. We will investigate how often this and other false negatives occur in section 6.2, and briefly suggest possible improvements to RMC to circumvent these issues when outlining future work in chapter 7.

Another weakness of RMC is the fact that it cannot deduce how many members of a varying set of machines to simulate, as was highlighted in the example above (section 5.1.1) when a message was broadcasted to the sharers vector. In our testing, we find that this does not have an impact in practice for the MSI, MESI, and MOSI protocols. However, we conjecture that more advanced protocols may require a more complex strategy; for example, it may be necessary to simulate every possible size of the set (which is required to be bounded to make verification tractable anyway). We leave it to future work to investigate this in more detail.

Finally, and most importantly, RMC suffers from making one key assumption about the protocol: that each transaction is *encapsulated*. By this we mean that every possible behavior in that transaction, i.e. all the different messages being sent between the machines, can occur without outside interference from another simultaneous transaction. This is because RMC cannot use auxiliary state to distinguish between which behaviors are possible in transaction-atomic (definition 3.2) systems and which are not. As we will see in section 6.2, this may cause false positives in some protocols, particularly those with fully unordered interconnects.

5.2 Identifying State Bugs with State Consistency Tracking

We have now seen how we can use Reduced Model Checking to catch message bugs and a small subset of state bugs, but what about the rest of the state bugs? As always, falsification engines such as Murphi can produce counter-examples showing how they break the system, but can we design a more fine-grain method of finding state bugs? To do this, we start with one key observation of the nature of directory-based cache coherence protocols:[2]

> **Observation 5.2.** In a directory-based cache coherence protocol, **the directory serves to summarize the state of the caches**.

This observation follows from the core motivation behind using a directory: it gives a readily available summary of the system state, so that communication does not need to be broadcast to all of the machines taking part in the system. As such, taking this observation to heart should not seem too far fetched; if we do, we can make an even stronger observation of the nature of directory-based cache coherence protocols:

> **Observation 5.3.** Since the directory summarises the state of the caches, any well-formed directory-based cache coherence protocol implicitly defines a **function** which maps **from the state of the caches to the state of the directory**. In other words, if a certain state of the caches is observed to occur paired with two distinct states of the directory, the protocol is not well-formed.

> **Remark.** Does this last observation (5.3) really hold true in all directory-based cache coherence protocols? If we consider MSI, MOSI, and MESI (as defined by Nagarajan et al. (2020), the go-to textbook in the field), we find that it holds true only for the first two; it does not naively hold for MESI. However, MESI can very easily, and in a way which is completely equivalent in terms of safety and performance, be adjusted so that it does satisfy this observation. We invite the

[2]To clarify: here we take *state* to include both the explicit state (e.g. M, S, I) and the implicit state (e.g. sharers vector).

```
Algorithm 4: State Consistency Tracking (high-level overview)
Result: Success, or a StateMismatchError
$F$ := a map from states of the caches to the directory state, initially empty ;
$S$ := the initial system state ;
while there are unexplored system states do
    $C$ := the state of the caches in $S$ ;
    $D$ := the state of the directory in $S$ ;
    if $C$ is not in $F$ then
        $F[C] := D$ ;
    if $F[C] \neq D$ then
        return StateMismatchError ;
        ;▷ The user should now be provided with one trace showing
          (C, F[C]) and one showing (C, D).
    $S$ := some previously unexplored system state ;
return Success ;
```

curious reader to read how in appendix E.

This observation, though deceptively simple, is very powerful since it gives us an algorithmic way of detecting many state bugs: simply explore the system state space, building this function as we go along and throwing an error if we ever find a new directory state to be paired with an existing cache state. This is precisely the algorithm which we dub State Consistency Tracking (SCT). It is noteworthy that upon encountering an error, SCT must provide the user with *two* error traces: one being the trace where it first learned the value of the function, and one where it found a mismatch.

Unlike Reduced Model Checking, SCT is easy (and we believe illustrative) to describe in abstract terms; algorithm 4 does just this. All the same, we believe it to be helpful to also look at a concrete example of applying SCT to verify a protocol, similarly to how was done for RMC in the preceding section. Since walking through the verification of a full protocol would be very long (and dull) and there is no sense of verifying a single transaction with RMC, we instead consider two individual examples of finding bugs in tests from the bug suite (chapter 4).

5.2.1 Example: Finding a State Bug in the Directory with State Consistency Tracking

For this example, consider the very first protocol in the bug suite introduced in chapter 4. The protocol matches MSI precisely, with the exception that when the directory is in state `I` and receives a `GetS` message, it forgets to add the requesting cache to the sharers vector. Clearly this is a state bug under our definition, and it is also one which can be found with SCT.

To find a bug with SCT, all we need to do is to find two system states which are both reachable from the initial state, and in which the cache states match but the directory states do not. In this case, doing so is relatively easy: figure 5.3a shows that by alternating whether one of the caches initially executes a load or a store, the mismatch can be found.

This example reveals one problem with SCT: it cannot tell us *which* of the two directory states found is the correct one. In fact, if there are multiple bugs present, it is possible that neither of them are. All SCT tells us that it cannot be that they are both correct, or equivalently, that at least one of them is wrong. When using SCT, care must thus be taken to analyse both candidate traces.

5.2.2 Example: Finding a State Bug in the Caches with State Consistency Tracking

The preceding example is a textbook example of when SCT works best, in that the bug actually lies in the directory. What happens if the bug lies in the caches? Luckily, SCT can still detect bugs present in the caches. To see this, consider protocol number 5 from the bug suite in chapter 4. Here the MSI protocol is incorrectly specified so that when a a cache receives the acknowledgement indicating that it may go ahead with its `I` $\rightarrow$ `S` transition, it forgets to update its explicit state to `S`. This causes a mismatch with the initial state of the system, as is shown in figure 5.3b.

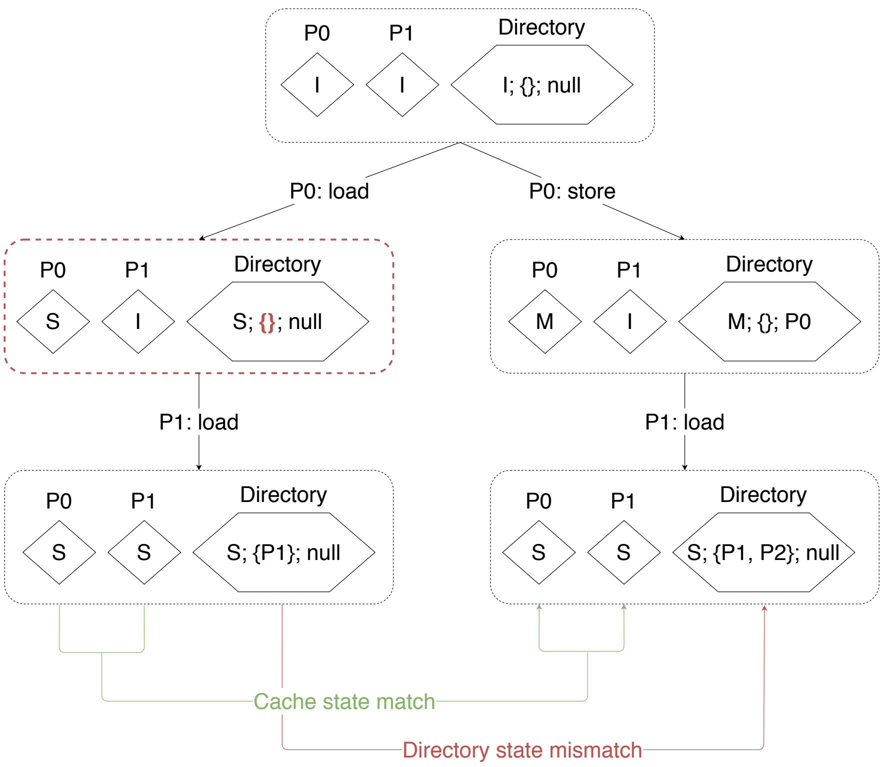
This second example shows another fundamental problem with SCT: even if it is the caches that are at fault, it is the directory that is in some sense blamed. This could be remedied if we expected the function from the cache states to the state of the directory to be bijective. However, it is very easy to imagine protocols in which this is not true: for example, if the directory does not store a sharers vector and instead broadcasts invalidations (but still maintains an owner pointer). Thus, when using SCT, one must keep in mind that the bug does not necessarily lie in the directory.

5.2.3 Strengths and Weaknesses of State Consistency Tracking

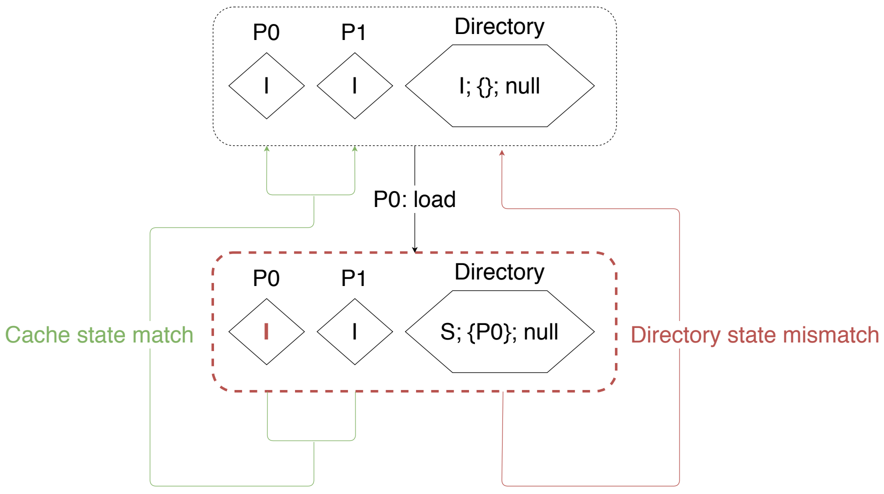
In the two preceding examples, we saw how SCT can be used to find bugs in stable state protocols. However, we also saw that just like RMC, SCT has its shortcomings. In this section, we will discuss in more detail when we expect SCT to be useful, and when we expect it not to be (again deferring quantitative evaluation to chapter 6).

The biggest strength of SCT, and the key reason why we expect it to be a useful debugging aid, is that (as was discussed in section 4.2.2) state bugs often lead to errors much later than the transaction in which the error actually occurred. With SCT, an error can be thrown as soon as the offending transaction has finished. Thus, the amount of information which the user is presented with but which really does not help them find the bug is potentially reduced, even more so if we have already used RMC to make sure that the cause of the bug is not the message flow itself.

Another strength of SCT is that along with reducing the amount of irrelevant information presented to the user, it may increase the amount of *relevant* information presented by giving the user an example trace where the directory state was different. This trace may reveal more closely where the issue lies; for example, inspecting the two traces



(a) Finding a state bug in the directory (section 5.2.1).



(b) Finding a state bug in the caches (section 5.2.2).

Figure 5.3: Using State Consistency Tracking to find bugs in the examples from sections 5.2.1 and 5.2.2. The directory state is given as "{explicit state}; {sharers vector}; {owner pointer}". The offending transaction, and the incorrect state updates therein, in bold and red. Note that for simplicity the auxiliary state in the caches is not included in these diagrams, although it is taken into account to form the state of the caches.

(paths down the tree) in figure 5.3a one can deduce that the bug does not lie in the update of cache state when a cache faces a load in I, since this is present in both traces.

Compared to RMC, we believe the drawbacks of SCT to be more severe. In the preceding sections we saw two such drawbacks: it does not answer the question of which (if any) of the two traces is the correct one, and it will point towards the directory even if it is the caches that are at fault. There is however an even more severe issue: to find a mismatch in directory state, we need to find two different traces which end up in the same state of the caches. This may not seem like too much of a problem in theory, given how highly connected cache coherence protocols are. It is however a problem in practice, because when there is a state bug in a protocol certain transitions in the protocol may very well deadlock, meaning they cannot be travelled down. In our implementation of SCT we try to avoid this problem by disregarding transitions which have been found to cause deadlocks when exploring the state space, but this of course means that SCT will not be able to find the bug if it lies within such a transition. We will see in section 6.3 that many state bugs fly under SCT's radar for this reason.

Another issue with our implementation of SCT is that to avoid having to conquer the very challenging task of building a full on model checker from scratch (or modifying the complex codebase of Murphi), we implement SCT by iteratively invoking Murphi, parsing its output, and adding invariants corresponding to the learned function as we go along. As Murphi is a compiled language, this means that SCT is potentially quite slow due to having to re-compile the Murphi output over and over again; for example, finding the bug discussed in section 5.2.1 takes about 30 seconds on a MacBook Pro (13-inch, early 2015). As we will see in section 6.4, this makes SCT run for a very longer time in exceptionally rare cases. However, this is not an inherent issue with the SCT algorithm; it is clearly not more complex than the original model checking task. Rather, it is an issue with our implementation of SCT, and one which could be resolved by natively implementing SCT into a model checker.

5.3 Bringing It All Together: Developing Stable State Protocols

We have now defined a SSP correctness criterion and derived two alternative methods for finding bugs in SSPs. Before evaluating these in the next chapter, let's round off our discussion by bringing it all together and outlining our vision of what the development process for directory-based cache coherence protocols may look like with the tools and insights laid out in this dissertation.

The old development process, using purely ProtoGen as presented by Oswald et al. (2018), is shown in figure 5.4. This process is a vast improvement over constructing and verifying the entire concurrent protocol from scratch, but it is still challenging for the designer. When faced with a counter-example claiming their designed protocol to be incorrect, the designer first has to parse the (potentially long and complex, including high degrees of concurrency) trace to figure out what is causing the safety violation. Then, they have to deduce whether the bug lies in a concurrent transient state generated by ProtoGen (indicating that ProtoGen has introduced the bug), or whether it is truly

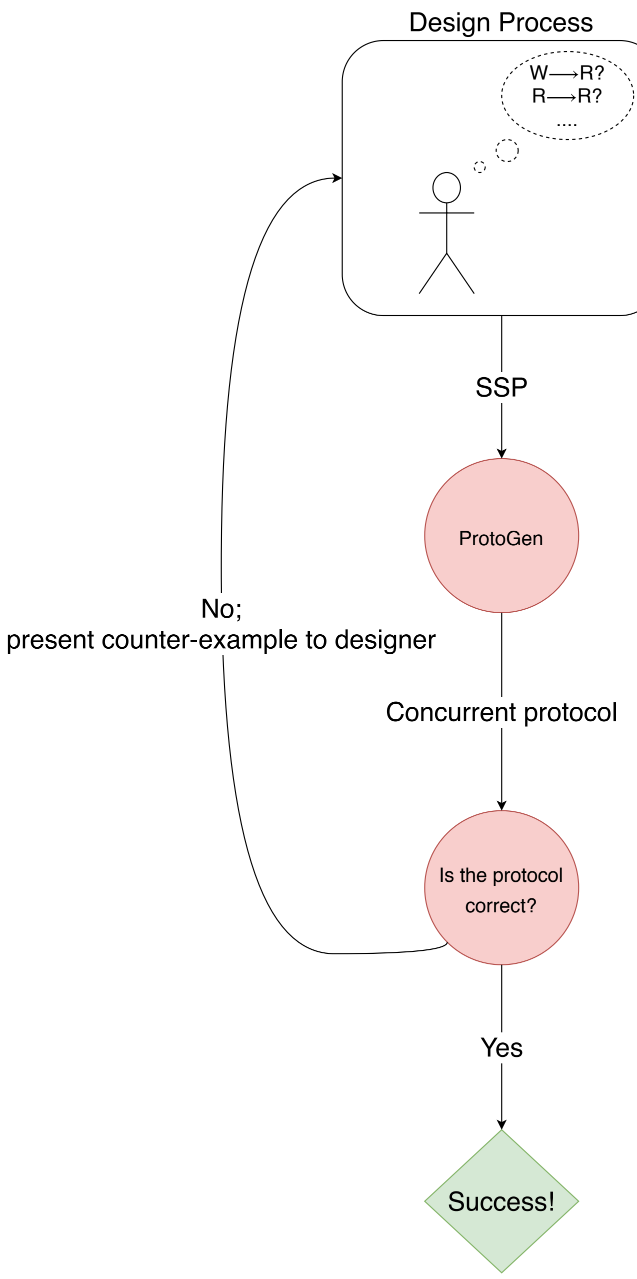


Figure 5.4: The cache coherence protocol development process before this dissertation.

present in their design. Only having completed these two cumbersome steps can they revisit their design and restart the process.

The new development process as envisioned by us is shown in figure 5.5. This takes full advantage of the tools presented in this dissertation. First, the SSP is verified against the correctness criterion (definition 3.3); this way, the designer does not have to worry about whether the bug lies in a concurrent state generated by ProtoGen. Then, if there is a bug present, Reduced Model Checking is carried out on the protocol to see if the bug can be located in the message flow. Failing this, the slower (and, as we will see in chapter 6, less accurate) State Consistency Tracking method tries to find an inconsistency in the protocol. If even this fails, the process simply falls back on the original workflow.

We hope that this section, albeit short, has helped the reader gain a more thorough understanding of the motivations behind the work laid out in this dissertation, and how it fits into the greater picture of cache coherence protocol design.

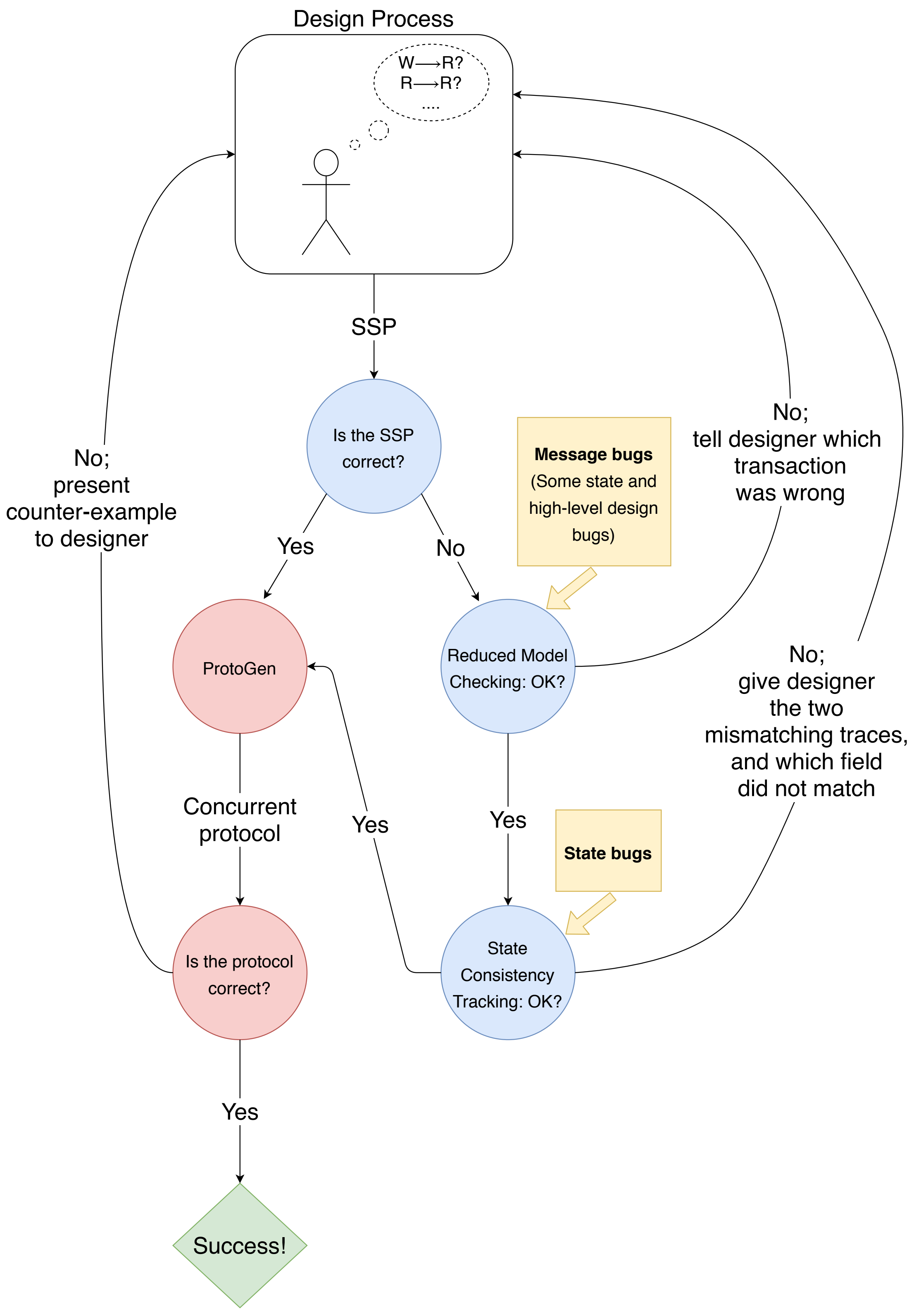

Design Process
W⟶R?
R⟶R?
....
SSP
Is the SSP correct?
Yes
No
Message bugs
(Some state and high-level design bugs)
No;
tell designer which transaction was wrong
No;
present counter-example to designer
ProtoGen
Reduced Model Checking: OK?
No;
give designer the two mismatching traces, and which field did not match
Concurrent protocol
Yes
Yes
State bugs
Is the protocol correct?
State Consistency Tracking: OK?
Yes
Success!

Figure 5.5: The cache coherence protocol development process using the tools and insights presented in this dissertation. Old steps in red, steps introduced by this work in blue. Yellow notes clarify which bugs are targeted at what stage of the verification process.

Chapter 6

Evaluation

In this dissertation we have presented the following work:

1. In chapter 3 we defined a criterion for SSP correctness (definition 3.3) and discussed how to implement this into a model checker

2. In chapter 4 we constructed a dataset of buggy protocols, and derived from it a taxonomy of the types of bugs which we find to appear in cache coherence protocols

3. Finally, in chapter 5 we introduced two novel methods for identifying those bugs which belong to the message or state classes: Reduced Model Checking and State Consistency Tracking

Before rounding things off in chapter 7, we will now evaluate the work presented in the preceding chapters. We begin by showing that the SSP criterion leads to error traces which are shorter, simpler, and expose the user to less transient states (section 6.1). We then move on to quantitatively evaluating the usefulness and coverage of the Reduced Model Checking and State Consistency Tracking techniques. We devote most of our time to evaluating them in isolation in sections 6.2 and 6.3, but as a sanity check we then also briefly evaluate the full development process (as described in section 5.3) in section 6.4. Finally, we round off the chapter by discussing shortcomings of our evaluation procedure in section 6.5.

Note that throughout this chapter, when invoking the Murphi model checker we use its breadth-first mode, allocate 2MB of memory to the hash tables, and consider a system with 3 caches; these are all standard settings in ProtoGen.

6.1 Usefulness of the SSP Correctness Criterion

Without further ado, let us begin the meat of this chapter by evaluating how useful the SSP correctness criterion (definition 3.3) really is.

As discussed in chapter 3, the main motivation behind defining a correctness criterion for SSPs is to reduce the number of auto-generated transient states which the user has to reason about, making the debugging process easier. However, we also hypothesise that it can make the error traces easier to reason about by shortening them and reducing the number of caches involved in the traces.

To evaluate the truthfulness of these two claims, we run ProtoGen on each of the 31 test protocols found in the manually created bug suite; once with SSP verification enabled, and once without it. The full results are presented in appendix C; in summary, we find that for 26 of the protocols Murphi is able to generate error traces both for the SSP and the full protocol. We will form the basis of our evaluation of the usefulness on these 26 protocols, since they allow us to quantitatively compare the error traces. We then take the error traces for these 26 protocols and gather three statistics on them, with and without SSP verification:

1. The number of unique transient states encountered over all of the traces;
2. The mean number of transitions in each error trace;
3. The mean number of caches involved in the error trace.

The results are presented in table 6.1. These confirm our hypotheses: when the SSP verification criterion is used, the user is exposed to less transient states and has on average roughly 2 less transitions to worry about, and the error traces involve less caches firing accesses. We remind the reader that the reason that any transient states are encountered at all with the SSP verification procedure is because we allow for the transient states which simply implement the atomic transitions of the SSP, as was discussed in section 3.1. We would also like to point out to the reader that we find that the SSP verification criterion is in fact even more helpful than this data may suggest: the transaction-atomicity of the system reduces the cognitive load of parsing the error trace, since transactions can easily be considered one at a time.

Statistic	Without SSP verification	With SSP verification
Unique transient states encountered	13	9
Mean number of transitions	9.73	7.58
Mean number of caches	2.58	1.81

Table 6.1: Summary quantitative results for the evaluation of the SSP criterion (lower is better for all). ”Without SSP verification” refers to verifying the protocol in Murphi after running it through ProtoGen, ”With SSP verification” refers to doing so in a transaction-atomic system.

As mentioned earlier, there are 5 test protocols (number 15, 17, 18, 20, and 28; see appendix C) from the bug suite which do not lend themselves nicely to quantitative comparison because the Murphi model checker fails to produce an error trace either with or without SSP verification. However, these protocols still reveal critical qualitative insights into the usefulness of the SSP criterion.

When verifying test protocol 15 without the SSP criterion, the Murphi model checker ran out of space before being able to find any bugs, which did not happen when the SSP criterion was used. This showcases another strength of the SSP criterion which we have thus far glanced over: reducing the verification state space. When the system model is made transaction-atomic, it greatly reduces the number of combinations of cache states which are possible since no two caches can be in transient states at the same time. Thus, it allows the model checker to handle test protocol 15 despite its bug blowing up the state space. We believe this aspect of the SSP verification procedure could prove very useful when verifying large system models, i.e. those with a large number of states and many caches.

Test protocol 20 is interesting because without the SSP criterion, it is not faulty at all. Recall from chapter 4 that this test removed the declaration that `Inv_Ack` messages (invalidation acknowledgements) may arrive before the `GetM_Ack_AD` has. When we verify this protocol without the SSP criterion, it passes verification because premature `Inv_Ack` messages can simply be stalled until the `GetM_Ack_AD` acknowledgement has arrived. However, with the SSP criterion we disallow all stalling, and hence such premature `Inv_Ack` messages instead lead to an error being thrown. We believe this to be of benefit to the user, as it alerts them that they have missed a possible optimization in the protocol.

Finally, test protocols 18 and 28 are, as mentioned in chapter 4, *concurrency bugs*. As such, they do not show up as errors in the SSP verification process, a flaw which we discussed at greater length in section 4.2.4. Nonetheless, the existence of these bugs does not invalidate the usefulness of the SSP criterion for other types of bugs, as long as the user is aware of their existence.

6.2 Reduced Model Checking

We now turn our gaze towards Reduced Model Checking (RMC), a method which we introduced in section 5.1 for finding message bugs. We believe the primary strength of this method is the fact that it considers each transaction (recall definition 3.1) individually. This means that when RMC finds a bug, the user knows precisely which transaction it occurred in; in fact, RMC can sometimes tell them precisely which message is faulty, for example if it is being sent to the wrong machine. If it is the case that the error traces generated by Murphi for message bugs typically contain more than one transaction, or in other ways make the user consider information which is not really relevant to the bug itself, then RMC is helpful to the user as it allows them to narrow their focus during the debugging process. This hypothesis is precisely what we set out to evaluate in this section. However, we also need to establish when RMC fails, i.e. when it gives false negatives or false positives, and how often this occurs.

These are questions which would hardly be adequately answered by merely evaluating RMC on the 4 message bugs found in our test suite. Therefore, we begin by deriving a way of automatically generating message bugs, so that we can produce a larger sample size.

6.2.1 Automatically Injecting Message Bugs

In our branch of ProtoGen, one can find a script called `MsgBugInjector.py`. This script automatically injects message bugs into a protocol by using the fact that from the definition of message bugs in section 4.2.1, we can deduce that message bugs can be injected by either mutating an existing message declaration, deleting one, or adding a completely new one. As such, these are the three modes supported by the script; each bug injected can use any of these modes, independently of the others. It should however be noted that the script makes some simplifying assumptions:

- When mutating messages, we mutate only the message type, its source field and its target destination.
- When adding messages, messages can only be added to transitions which are already sending out at least one message. This makes it easier for the script to inject the message into the file in such a way that it is still well-formed, i.e. message declarations are not inserted in a section of the input which does not describe some part of the finite state machines.
- Each expression in the file is contained in one line. This makes the task of parsing the file significantly easier.

Table 6.2 describes some example buggy protocols which the message bug injector may generate. The last one of these is particularly interesting: the cache tries to send a writeback to the source of a `PutM` message, but in the current transition no `PutM` has been received. One might expect this to lead to a static error, but as it turns out, running ProtoGen on this protocol leads to a run-time error where the writeback is sent to the source of the `Fwd_GetS` (which the directory has spoofed to be the cache trying to obtain read permission). This is because ProtoGen's Murphi backend does not generate Murphi code which precisely matches that of the input. Whenever a field of a message is referred to, such as "`PutM`.src" or "`PutM`.dest", the Murphi code generated instead translates these to refer to the last message received in this transition, regardless of its type. With our in-house RMC, we can easily check for such errors by comparing the message type to that of the one most recently received.

6.2.2 Accuracy and Usefulness of RMC

Having derived a script which can inject arbitrary message bugs into a protocol, we are now prepared to investigate RMC in more detail. We begin by generating 300 (not necessarily unique) buggy protocols, for now only injecting 1 bug into each protocol. To get as much test coverage as possible, we generate 100 protocols based on MSI, 100 based on MESI, and 100 based on MOSI. We also split the types of message bugs evenly over adding, deleting, and mutating messages.

Despite ProtoGen's Murphi backend being insensitive of message names when referring to messages which have been received, it is still possible to inject bugs in such a fashion that the generated Murphi output will not compile. Out of the 300 protocols

Type of message bug injected	Machine	State	Received Message or Event	Description
Addition	Directory	S	GetS	In addition to GetS_Ack, send Put_Ack to source
Deletion	Directory	S	GetM	No longer broadcast Inv messages to sharers
Mutation	Cache	M	Fwd_GetS	Send writeback to ”PutM.src” instead of the directory

Table 6.2: Some examples of buggy protocols generated by the message bug injector.

we generate with the message bug injector, 77 fail to compile. Inspecting these in more detail, it becomes clear that there are two possible causes why a protocol would fail to compile. The first is that a message is added or mutated at the start of a store, load, or eviction transition, such that the cache now tries to refer to a field (e.g. the source) of a supposedly received message, but *no* messages have been received yet. In this case the generated Murphi code will refer to a variable which does not exist in the scope, and hence a static error is thrown. The other possible cause is that a message is deleted such that it is never sent anywhere; for example, removing sending invalidations when a directory in `S` receives a `GetM`, since this is the only time invalidations are sent. In this case, we find that ProtoGen does not declare this message type to exist in the generated Murphi file, and yet it will still try to match messages against this type, for example when caches in `S` receive messages. This causes a static error to be thrown, and Murphi will halt compilation.

When a protocol fails to compile in Murphi, we can still run RMC on it and find bugs; however, quantifying the usefulness of this is difficult and would require more sophisticated measures than we consider here. The protocols where Murphi cannot find an error (due to ProtoGen’s Murphi backend being insensitive of message names when accessing fields of received messages, as was just discussed in section 6.2.1) are equally difficult to evaluate our efforts on. Instead, we base our evaluation of RMC solely on the 214 remaining protocols which successfully compiled and which Murphi produced an error trace for.

We are now ready to quantitatively analyse RMC’s coverage. We find that out of the remaining 214 protocols, 13 false negatives (i.e. protocols where RMC does not flag any issues) occur. Table 6.3 summarises the results of our evaluation efforts thus far.

Having seen that RMC is able to find bugs in a vast majority of cases, we then seek to evaluate how useful it is when it does find a bug. To do so, we begin by noting down the number of transactions found in the error traces for the 201 protocols in which RMC flagged an error, when run through Murphi in a transaction-atomic system model. We find that the mean number of such transactions is 2.16. Recalling that RMC only ever

Group	Number of protocols
Failed to compile	77
Murphi found no error	9
RMC true positive	201
RMC false negative	13
Total	300

Table 6.3: Summary quantitative results for the evaluation of RMC's coverage.

considers one transaction at a time, this means that RMC reduces the mean number of transactions which the user has to consider when debugging their protocol by 1.16, over half of the original mean error trace length. However, this does not quantify the fact that when a bug is found with RMC, there is less information which needs to be analysed for each transaction since we know the source of the bug to be the message flows (or, in rare cases, a lacking update of state in the very last transition of one of the participating machines). In other words, when considering the error traces generated by Murphi one would need to look at both the messages and the update of state in the entire trace, but when RMC finds the bug this is not necessary; we often know precisely where the bug lies. In an effort to quantify how useful this is, we introduce the concept of a *complexity measure*, a primitive way of estimating how much information there is to consider when a bug is found through either RMC, SCT, or just plain Murphi. Its detailed specification is given in appendix F, but intuitively this seeks to capture in how many transitions one would need to look at either the message flow, the update of state, or both. Comparing then the complexity measures of the error traces generated by Murphi and the errors generated by RMC shows that RMC vastly reduces the amount of information to consider; these results are summarised in table 6.4.

Error found by	Mean number of transactions	Mean complexity measure
Murphi	2.16	14.18
RMC	1.00	3.32

Table 6.4: Summary quantitative results for the evaluation of RMC's usefulness. Calculated over the 201 protocols for which both RMC and Murphi found an error.

We have now evaluated RMC's coverage and usefulness on a suite of 300 randomly generated message bugs. However, the keen reader may argue that evaluating the number of false negatives given by RMC means little without also evaluating its rate of false positives (i.e. how often it flags correct protocols as being incorrect), which cannot be done when only considering buggy protocols. Unfortunately, we do not have access to large amounts of correct protocols to use for an extensive experiment. Our analysis is therefore limited to the MSI, MESI, MOSI, and MESI_unordered (like MESI but with a fully unordered forward interconnect) implementations which ProtoGen provides as sample protocols in it source code (ICSA-Caps, 2018). We find that for MSI, MESI, and MOSI, RMC does not throw any errors. However, in MESI_unordered it does. This is because, as discussed in 5.1.2, RMC assumes that each transaction is encapsulated: every message which *can* be sent by a machine should be able to be received by the target in at least one of the states that it can be in without any no outside interference. This assumption does not hold true for ProtoGen's sample MESI_unordered implementation, because it uses auxiliary state (ignored by RMC) to differentiate race cases which only arise when the forward network is unordered. In one of these, the directory may send a message to a cache which it can only receive if it has since sending the message been interfered with by an outside cache (which the directory deduces has happened from its auxiliary state). Since RMC is ignorant of this outside interference, it incorrectly reports that there is an error. We recognise this aspect of RMC as one of its core limitations, as fully unordered interconnects are not uncommon in cache coherence protocols.

To conclude this section, we consider what would happen if there were multiple message bugs present in the protocols. Luckily, since RMC considers each transaction individually, bugs present in other transactions can by construction not prevent it from finding a bug in the one currently being considered. We do hypothesise it to be possible that having several bugs within the same transaction can cause these bugs to cancel out in a way such that if only one of them were present it would be found by RMC, yet when they are all present RMC is incapable of finding it. However, despite our best efforts, we have not been able to produce such an example based on any of the MSI, MESI, or MOSI protocols. We leave it to future work to investigate this hypothesis on more advanced cache coherence protocols, such as TSO-CC (Elver and Nagarajan, 2014).

6.3 State Consistency Tracking

Having evaluated RMC's suitability as a method for finding message bugs, we now move on to evaluating State Consistency Tracking (section 5.2) and its ability to detect state bugs. We hypothesise that SCT can detect state bugs before they cause safety or deadlock violations, therefore making the error traces shorter. However, we also hypothesise that SCT is relatively sensitive to false negatives, as discussed in section 5.2.3.

To confirm these hypotheses, we will quantitatively evaluate SCT's performance on a large set of buggy protocols. Just like with RMC, we therefore begin by deriving an automated way of generate state bugs so that we can get a larger sample size than we could from the bug suite.

6.3.1 Automatically Injecting State Bugs

The state bug injector can be found in `StateBugInjector.py` in the source code of our branch of ProtoGen. It supports the same three modes of bug injection as the message bug injector: adding state updates, mutating existing ones, and removing existing ones. Just like the message bug injector, it assumes that every expression in the input file is contained in a single line, and it only adds state updates to transitions which are already updating some state. Furthermore, when adding state updates, it only ever adds updates to state that actually belongs to the machine; this prevents the script from generating very unlikely bugs, such as attempting to manipulate the sharers vector in a cache which does not have such a construct present.

Table 6.5 lists some example state bugs generated by the script. Just like with the message bug generator, the state bug generator suffers from some weirdness due to how ProtoGen's Murphi backend aliases messages names. This is for example shown in the first of the bugs in table 6.5; just like before, this bug does not cause a static error, but instead will add the source of the PutM message to the sharers vector.

Type of state bug injected	Machine	State	Received Message or Event	Description
Addition	Directory	M	PutM	Add "GetM.src" to sharers vector
Deletion	Cache	IM	Inv_Ack (before GetM_Ack_AD)	Deleted increment of acknowledgements counter
Mutation	Directory	S	PutS	If the sharers vector is now empty, go to M (instead of I)

Table 6.5: Some examples of buggy protocols generated by the state bug injector.

6.3.2 Accuracy and Usefulness of SCT

Similarly to how was done for RMC, we begin by generating 300 protocols containing one single state bug. We split these evenly over the modes of operation of the state bug injector and the base protocols (MSI, MOSI, and MESI[1]).

Just like with RMC, some of the generated protocols will not compile in Murphi due to referring to contents of a message despite no message being in scope. However, this is a much rarer case with the state bug injector, because of factors such as a majority of state updates taking place in the directory (whose transitions always begin with receiving a message). We find that only 1 out of the 300 protocols generated fail to compile under Murphi.

Five of the generated protocols do not make it past ProtoGen at all, either due to breaking ProtoGen's own parser or because they cause the ProtoGen algorithm to get stuck in an infinite loop[2]. Another eight protocols break static checks such as state reachability (see chapter 3). Finally, 140 of the generated protocols in fact pass as correct. This is because they modify state in a way which only breaks the assumption that the directory state is a function of the state of the caches; this is required by SCT but is *not* required by the SSP correctness criterion (definition 3.3), so they pass verification.

In the end, we are left with 146 buggy protocols to carry out our quantitative analysis of SCT on. Out of these, we find that 35 fly under SCT's radar because their bugs lie

[1]Here we use the modified MESI protocol, for which observation 5.3 holds. See appendix E.

[2]The protocols which completely break ProtoGen's algorithm have been supplied to Nicolai Oswald, author of ProtoGen, for review.

within transitions which themselves cause deadlocks, as discussed in section 5.2.3. In the remaining 111 protocols, SCT finds a state inconsistency; we find this coverage acceptable, but noticeably worse than that of RMC (201/214 protocols). Table 6.6 summarizes our findings thus far.

Group	Number of protocols
Failed to compile	1
Broke ProtoGen	5
Broke static checks	8
Murphi found no error	140
RMC true positive	111
RMC false negative	35
Total	300

Table 6.6: Summary quantitative results for the evaluation of SCT's coverage.

When SCT does find an inconsistency, we record the lengths (number of transactions) in each error trace (the original error trace and the two mismatching traces provided by SCT). Just like with RMC, we do however find that this does not adequately quantify the usefulness of SCT. When following the development process suggested in section 5.3, that is when SCT is only ever invoked when RMC is unable to find a bug, we can be quite certain that if SCT finds a mismatch it is because of the update of state in the protocol, not because of the message flows. We therefore again make use of the complexity measure specified in appendix F, collecting its value for each of the error traces. The results are shown in table 6.7. This shows that the two error traces generated by SCT are on average shorter than the original error trace generated by Murphi, but it also shows that their combined length is in fact longer! However, the complexity measure indicates that even if both traces need to be scanned completely, the fact that we know the bug lies in the state changes means that the total complexity measure is lower (averaging $8.65 + 4.36 = 13.01 < 19.65$).

Trace	Mean number of transactions	Mean complexity measure
Original	2.95	19.65
Longer mismatch	2.52	8.65
Shorter mismatch	1.43	4.36

Table 6.7: Summary quantitative results for the evaluation of SCT's usefulness.

What about false positives? Recall that since SCT only checks whether observation 5.3 holds in the protocols, it will not give any false positives as long as the protocol designer has ensured this holds throughout the protocol.[3] As was remarked in section 5.2, we find this easily holds for the MSI and MOSI protocols, and that the MESI protocol (including MESI_unordered, which caused issues for RMC) can very easily be adjusted to support this. Thus, we do not find false positives to be a problem for SCT.

[3]For example, they must have ensured that the owner pointer is reset to null when the system transitions away from some core having read-exclusive permission to the line.

When there are multiple bugs present, SCT will only locate one of them at a time. However, having multiple bugs present may inhibit SCT's ability to find any bugs at all. Unlike for RMC, it is easy to construct protocols in which several bugs cancel each other out so that SCT cannot find them. Intuitively, SCT cannot find inconsistencies if the protocol is *consistently wrong*. Consider for example an MSI protocol where the designer forgets to ever update the sharers vector: it is always empty. Such a protocol will clearly break the Single-Writer-Multiple-Readers invariant as the protocol is effectively incapable of demoting sharers, but SCT will not be able to find any inconsistencies as the protocol is consistently wrong. Another problem with increasing bug counts is that as the number of state bugs present in a protocol goes up so does also the number of deadlocked transitions, and the more deadlocked transitions there are the harder it will be for SCT to traverse the state space enough to catch inconsistencies. At the same time, more bugs means that it is more likely one of them will be close to the initial state. Thus, it may not be a problem for SCT in practice that it may not be able to traverse as much of the state space.

We finish off our evaluation of SCT by investigating how this trade-off works out in practice. To do so, we generate protocols containing 2, 4, or 8 bugs in addition to the single-bug protocols already generated. We set out to generate 180 buggy protocols, but as the number of bugs increases, so does the chance that a protocol will fail to make it past the static checks. To make sure the sample sizes do not get too small, we therefore rerun the process until for each number of bugs we have generated at least 60 protocols which make it through all of the static checks. We then run SCT on each of these protocols, and collect the ratio of true positives; the results are shown in figure 6.1. This shows that as the number of state bugs present increases, SCT is in fact increasingly capable of finding *some* bug.

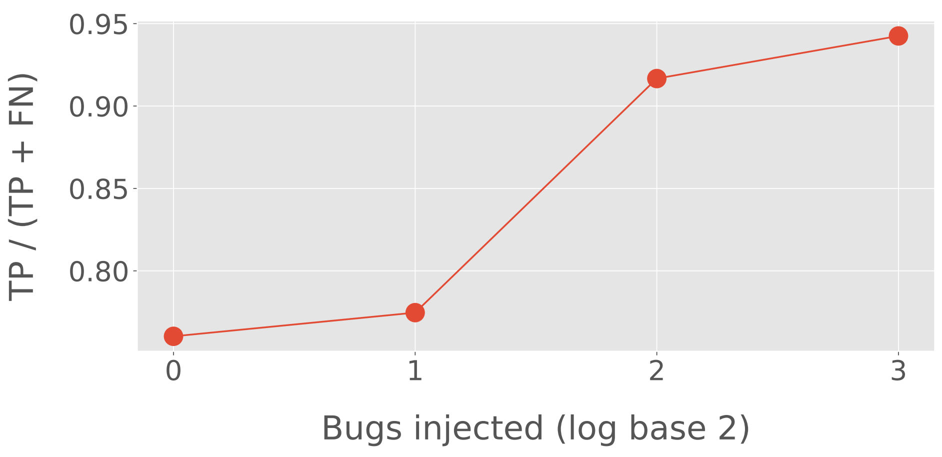


Figure 6.1: Ratio of true positives found by State Consistency Tracking as the number of state bugs injected into the protocol increases. Note that a true positive here is SCT finding *a* bug, not all of the bugs.

6.4 Using Reduced Model Checking and State Consistency Tracking Together

As was discussed in section 5.3, we suggest that Reduced Model Checking and State Consistency Tracking are seen as complementary methods, each focused on catching two different types of bugs. In this section, we briefly evaluate a full debugging system incorporating both methods on the bug suite from chapter 4, to make sure that using both methods in tandem does not lead to unexpected issues (e.g. the presence of state bugs does not mess up RMC). This debugging system first applies RMC and then carries out SCT only if RMC does not find any bugs, as shown in figure 5.5.

Because of its length, the table of test-by-test results is given in appendix D. In summary, we find that SCT misses two of the state bugs: 2 and 26. This is because in these two tests, the bug itself lies in a deadlocking transition; as discussed in section 5.2.3, SCT is unable to find inconsistencies in such cases. On the other hand, RMC misses one of the message bugs: 21. This is precisely the bug which was given as an example in section 5.1.2 for how RMC may miss some message bugs. On the other hand, each of SCT and RMC manage to catch two high-level design bugs each; this is testament to how high-level design bugs are perhaps not very well separated from the state and message bug classes, as pictured in the Venn diagram in section 4.3.

As discussed in section 5.2.3, our implementation of SCT is potentially very slow due to the need to compile Murphi code several times over. We find that for one of the protocols in the bug suite, number 10, SCT reaches our time limit of 10 minutes and is subsequently aborted. This is the only protocol considered in this chapter in which we have found SCT to take so long. Upon further inspection, we find that the reason that this protocol in particular leads to such a long run-time for SCT is that it causes the space of possible cache states to be very large by allowing for virtually infinite increments of the counter keeping track of how many `Inv_Acks` a cache has received. Thus, SCT constantly finds new cache states to add to its learnt function, each of which triggers a re-compilation of Murphi code; making the process very slow in total.

The above investigation shows that using RMC and SCT in tandem is an efficient way to cover many types of bugs in protocols. It also does not show any signs of the two methods interfering with each other. For example, the presence of state bugs does not appear to cause RMC to give false positives (as expected). However, it has also made clear that there are cases were other types of bugs, such as high-level design bugs, may have adverse effects on the methods.

6.5 Discussion: Shortcomings of the Evaluation Process

When discussing the validity of the bug suite and taxonomy in section 4.3, we mentioned that one common thread in this project is that we rarely have the luxury of relying on established practices and existing literature. This becomes most obvious in this evaluation chapter. Like a dictator who announces themselves judge, jury and executioner, we construct the methods to evaluate, the datasets to evaluate them on, and the metrics to evaluate them with. As a result, there are several criticisms one might raise about the evaluation process presented in this chapter. In this section, we aim to acknowledge and briefly discuss two such issues.

In our eyes, the most glaring concern with the evaluation process is that some of the metrics used to evaluate State Consistency Tracking and Reduced Model Checking, such as the mean number of transactions which need to be considered by the user, are perhaps not immediately tied to the usefulness of the method. At the same time, the more directly related complexity measure specified in appendix F may seem to have a rather arbitrary definition, and one might certainly have concerns regarding how accurate of a picture this measure really paints of the methods' usefulness. Ultimately, since the goal of this dissertation is to develop methods which can be used to guide synthesis, these quantitative measures were chosen for evaluation as an attempt at indicating what the complexity of the synthesis step would be following an error found by either method. If the core focus of the dissertation had instead been on designing more user-friendly tools, alternative evaluation processes such as user studies might have given clearer results. Nonetheless, we would like to point to the fact that much of the existing literature in this field, including both Teapot (Chandra et al., 1996) and TRANSIT (Udupa et al., 2013), were evaluated purely by presenting case studies of the authors using their own tools to construct some protocols. Indeed, Udupa et al. (2013) acknowledge that ”a direct scientific comparison with existing approaches is challenging”. We hope that the quantitative measures used to evaluate SCT and RMC in this paper, although not flawless, are a step in the right direction for this field.

Another concern is that the datasets used to evaluate the methods, including those automatically generated by the scripts described in sections 6.2.1 and 6.3.1, are ultimately subject to our biases and may therefore not accurately reflect the problem space in general. As discussed in chapter 4, these datasets are really a best effort to remedy the fact that there is no publicly available data to quantitatively evaluate our methods on, or to even use for inspiration in their design. As future work in this field unveils further insights into the types of bugs which appear in cache coherence protocols, we hope that the datasets can evolve with them.

Chapter 7

Conclusion

7.1 Summary

This paper has presented contributions towards the automatic synthesis of cache coherence protocols by investigating the directory-based Stable State Protocols (SSPs) taken as input by the existing synthesis tool ProtoGen (Oswald et al., 2018).

Chapter 3 proposed a concrete correctness criterion for SSPs, along with a way to verify this criterion in the Murphi model checker (Dill et al., 1992). The correctness criterion was argued to separate the correctness of ProtoGen from the correctness of the user's input, thus making localising the bugs easier. Quantitative evaluation thereof in section 6.1 showed that the criterion lead to error traces which are shorter in length and less complex than the ones obtained by naively using Murphi, exposing the user to less transient states and typically involving less machines.

Chapter 4 then discussed how cache coherence protocols may be incorrect, presenting what is to our knowledge the first publicly available dataset of faulty protocols and proposing a taxonomy of the bugs which appear in cache coherence protocols. One of the proposed classes of bugs, dubbed *concurrency bugs*, highlighted that there is a gap between when an SSP is correct per the SSP correctness criterion presented in chapter 3 and when ProtoGen can use the SSP to derive a full, correct protocol. This gap arises due to ProtoGen only being able to fill in holes in the design which arise due to races on the interconnects; it is not able to synthesise entire message flows in the protocol which allow it to safely handle concurrency. Section 4.3 concluded the chapter by discussing other shortcomings of the bug suite and taxonomy, acknowledging limitations thereof but also arguing for their usefulness in spite of this.

Finally, chapter 5 introduced Reduced Model Checking (RMC) and State Consistency Tracking (SCT), two novel methods for locating bugs in SSPs. RMC finds message bugs at the granularity of a single transaction by looking for inconsistencies in the message flows between the cache controllers and the directory controller. Quantitative evaluation of RMC in section 6.2 showed that it was able to find the bug in 201 out of 214 single-message-bug protocols. It was also shown that when RMC was able to locate a bug, it on average reduced the number of transactions which the user would need to consider and that the amount of information which the user would need to parser to locate the true cause of the error was drastically reduced. SCT, on the other hand, targets state bugs by finding inconsistencies in the manipulation of state in the protocol. Evaluation of SCT in section 6.3 showed that it found an inconsistency in 111 out of 146 single-bug protocols. When it did so, the two traces leading to the inconsistency were both on average shorter than the original trace generated by Murphi, and their sum total complexity was lower than the original trace due to knowing the fault lies in the update of state in the protocol.

7.2 Future Work

This paper has only laid the groundwork for future research towards synthesising fixes to bugs in SSPs, focusing on bug localization efforts which can guide the synthesis process. For the second part of the project, we suggest to consider the following directions of research:

- **Investigating the correctness of stable-state specifications of snooping protocols.** This work has focused exclusively on directory protocols, since this is the only class of protocols which ProtoGen can handle. Snooping protocols are inherently different from directory protocols, in that they contain less auxiliary state and have an implicit serialization point at the shared interconnect rather than at the directory. Investigating whether the SSP correctness criterion is suitable also for these protocols, and if so in what ways they may break the criterion, could reveal insights into the wider applicability and usefulness of the methods presented in this paper.

- **Extending Reduced Model Checking.** We suggest that RMC may be improved by storing information regarding which transitions in the NFAs have been taken at some point in the protocol, and which have not. Bugs which do not become obvious when simply simulating the NFAs could then be detected by noticing that they cause certain transitions to never occur in the protocol.[1] Consider for example the bug discussed in section 5.1.2, where a sharing cache forgets to respond with an `Inv_Ack` when invalidated. This would cause the ”Get `Inv_Ack`” transitions in the requesting cache's NFA to never be fired, at which point RMC could throw an error alerting the user of this.

- **Bringing State Consistency Tracking in-house.** Natively implementing SCT into either an existing model checker or one developed from scratch could circumvent the issue of it being slow to run due to having to re-compile Murphi code for each new cache state found, as discussed in section 5.2.3. This would make SCT more practically useful, and may also help with avoiding deadlocking transitions by making it easier to deduce precisely which transition is causing the deadlock.

[1]This idea was suggested to us by Nicolai Oswald of ProtoGen (Oswald et al., 2018).

- **Finding message bugs statically with session types.** Session types (see for example Dezani-Ciancaglini and de'Liguoro (2010) for a recent survey) is a type theory developed to be used for communication-centric programs, such as web protocols. Motivated by the intuition that RMC finds bugs by looking for inconsistencies in the message flows between the cache controllers and the directory controller, we believe session types could be a way to look for such inconsistencies in the message flows in a more formal and structured manner. In fact, we conjecture that RMC may be analogous to dynamic (i.e. run-time) type-checking of implicit session types, although we admit a much more rigorous investigation of RMC would be needed to see if this claim holds any weight in practice.

- **Using RMC and SCT to guide synthesis of bug-fixes.** This is the ultimate goal of our project, and the future work which we consider to be most interesting. Having seen that we can find bugs in SSPs at quite fine granularity, would it be tractable to synthesise fixes to bugs with simple brute-forcing? If so, what actions are made available to the synthesis engine, and how do we search for fixes? One challenge is that the actions available to the synthesis engine would have to be deduced from the user's input (since not all protocols are made up of the same messages and auxiliary state), and if the user's input is erroneous then it may not be possible to deduce the existence of the action (for example, a certain type of message) which would in fact resolve the bug. If brute-forcing synthesis would prove intractable, recent advances in program synthesis guided by machine learning (e.g. Zhang et al. (2018)) could be tried. However, techniques proposed in this field have to our knowledge not yet been applied to programs as complex as coherence protocols, and their application thereon would therefore likely not be straight-forward.

Bibliography

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44. Springer, 2011.

Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 261–274. IEEE Press, 2018. ISBN 9781538659847. doi: 10.1109/ISCA.2018.00031. URL `https://doi.org/10.1109/ISCA.2018.00031`.

[Arm Research]. Cache coherence protocols are notoriously ~~Hard~~ easy Prof. Vijay Nagarajan, Uni. of Edinburgh, October 2018. [Video File]. Retrieved February 2020 from `https://www.youtube.com/watch?v=ZfMyDDsN5u4`.

Graham Brightwell and Peter Winkler. Counting linear extensions is #p-complete. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 175–181, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897913973. doi: 10.1145/103418.103441. URL `https://doi.org/10.1145/103418.103441`.

Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 237–248, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917952. doi: 10.1145/231379.231430. URL `https://doi.org/10.1145/231379.231430`.

Satish Chandra, James R Larus, Michael Dahlin, Brad Richards, Randolph Y Wang, and Thomas E Anderson. Experience with a language for writing coherence protocols. In *Domain-Specific Languages*, pages 51–66, 1997.

Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in the disk based murϕ verifier. In *International Conference on Formal Methods in Computer-Aided Design*, pages 202–219. Springer, 2002.

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In Cosimo Laneve and Jianwen Su, editors, *Web Services and Formal*

Methods, pages 1–28, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14458-5.

D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 522–525, Oct 1992. doi: 10.1109/ICCD.1992.276232.

M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan. VerC3: A library for explicit state synthesis of concurrent systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1381–1386, 2018.

Marco Elver and Vijay Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 165–176. IEEE, 2014.

Brian T Gold. Teaching cache coherence protocols with model checking, Jan 2009. URL https://users.ece.cmu.edu/~bgold/teaching/coherence.html. Retrieved April 2020.

ICSA-Caps. ProtoGen source code, 2018. URL https://github.com/icsa-caps/ProtoGen. Retrieved August 2019. Version 0.4.

Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Trans. Archit. Code Optim.*, 11(4), December 2014. ISSN 1544-3566. doi: 10.1145/2663345. URL https://doi.org/10.1145/2663345.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*, volume 15. Morgan & Claypool Publishers, 2nd edition, 02 2020. doi: 10.2200/S00962ED2V01Y201910CAC049.

Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA ’18, page 247–260. IEEE Press, 2018. ISBN 9781538659847. doi: 10.1109/ISCA.2018.00030. URL https://doi.org/10.1109/ISCA.2018.00030.

D.L. Dill R. Melton. *Murphi Annotated Reference Manual*, release 3.1 (updated by C. N. Ip and U. Stern) edition, July 1996. URL https://www.cs.ubc.ca/~ajh/courses/cpsc513/assign-token/User.Manual.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174. URL https://doi.org/10.1145/2499370.2462174.

Patrick Valduriez. Shared-memory architecture. In LING LIU and M. TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2638–2638. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_1082. URL https://doi.org/10.1007/978-0-387-39940-9_1082.

Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*, pages 1737–1746, 2018.

Appendix A

Messages Used in MSI, MOSI, and MESI

Throughout this dissertation, we have used the names of specific types of messages occurring in the protocols to discuss bugs. This index gives a complete explanation of these message types, so that the reader can refer to it if they at any point encounter a message they are not familiar with. Note that we use {X} as a placeholder for any appropriate state X. For example, Get{X} is shorthand for GetM and GetS.

Get{X}: Cache → Directory message indicating that the cache wishes to transition to state X.

Get{X}_Ack: Directory → Cache or Cache → Cache message indicating that the receiving cache is free to transition to X (unconditionally).

GetM_Ack_D: Directory → Cache or Cache → Cache message indicating that the cache can go ahead with its transition to M without waiting for invalidation acknowledgments (Inv_Acks).

GetM_Ack_AD: Directory → Cache message indicating that the cache can go ahead with its transition to M upon having received as many Inv_Acks as stated by the message's payload.

Inv: Directory → Cache message indicating that the cache should invalidate the line and send out an Inv_Ack to the cache wanting to transition to M.

Inv_Ack: Cache → Cache message acknowledging that the source of the message has received and acted upon an invalidation request.

Fwd_Get{X}: Directory → Cache message forwarding a request from another cache (e.g. forwarding a GetM to a cache currently in M).

Put{X}: Cache → Directory message telling the directory that the cache has evicted a line it previously held in state X.

Put_Ack: Directory → Cache message acknowledging a Put of any type.

Appendix B

Stable State MSI Protocol

In this appendix, we give the complete stable-state specification of the MSI protocol introduced in section 2.2.1. Table B.1 describes the cache controller, while table B.2 describes the directory controller. The columns list the states of the protocol, while the rows list the messages and actions which may be received in a stable state. Grey cells indicate that the particular combination of state and message/action is not possible. Explicit state updates are always given at the very end of the event, following a forward slash (e.g. "/ S" to indicate that the machine will now go to state S). Both tables are based on material presented by Oswald et al. (2018), adjusted to use the message names listed in appendix A.

	I	S	M
Load	Send GetS to Dir, get GetS_Ack / S	hit	hit
Store	Send GetM to Dir, get GetM_Ack_D or (GetM_Ack_AD and Inv_Acks) / M	Send GetM to Dir, get GetM_Ack_D or (GetM_Ack_AD and Inv_Acks) / M	hit
Eviction		Send PutS to Dir, get Put_Ack	Send PutM to Dir, get Put_Ack
Fwd_GetS		Send GetS_Ack to requester / I	Send GetS_Ack to requester / I
Fwd_GetM			Send GetM_Ack_D to requester and WB to Dir / I
Inv		Send Inv_Ack to requester / I	

Table B.1: MSI cache controller

	I	S	M
GetS	Add requester to sharers vector, reply with GetS_Ack / S	Add requester to sharers vector, reply with GetS_Ack	Add owner and requester to sharers vector, send Fwd_GetS to owner, get WB from owner, unset owner pointer / S
GetM	Set owner pointer to requester, reply with GetM_Ack_D / M	Reply with GetM_Ack_AD (payload = number of current sharers), Send Inv to all sharers, set owner pointer to requester, clear sharers vector / M	Send Fwd_GetM to owner, set owner pointer to requester
PutS		Reply with Put_Ack, remove requester from sharers vector [if sharers vector now empty / I]	
PutM			Reply with Put_Ack, unset owner pointer / I

Table B.2: MSI directory controller

Appendix C

Bug Suite Test Results for the Evaluation of the SSP Correctness Criterion

Test	Transitions (not SSP)	Transitions (SSP)	Caches (not SSP)	Caches (SSP)	Notes
1	6	6	2	2	
2	10	6	3	1	
3	6	6	2	2	
4	6	6	2	2	
5	10	6	3	2	
6	10	11	3	2	
7	15	5	3	1	
8	7	8	2	1	
9	12	11	3	3	
10	6	6	2	2	
11	11	7	3	2	
12	15	11	3	2	
13	8	9	2	2	
14	18	14	3	2	
15					Full protocol caused Murphi to run out of memory for the hash tables
16	7	7	2	2	
17					Static error

Continued on next page

Test	Transitions (not SSP)	Transitions (SSP)	Caches (not SSP)	Caches (SSP)	Notes
18					Concurrency bug
19	6	6	2	2	
20					Full protocol verifies (stalls premature Inv_Acks)
21	10	7	3	2	
22	10	7	3	2	
23	11	6	2	1	
24	9	9	2	2	
25	7	3	3	1	
26	12	8	3	2	
27	16	12	3	2	
28					Concurrency bug
29	7	7	2	2	
30	8	7	3	2	
31	19	6	3	1	

Appendix D

Bug Suite Test Results for RMC and SCT

Test	Bug type	Caught by	Notes
1	State	SCT	
2	State	Neither	
3	State [2x]	SCT	
4	State	SCT	
5	State	SCT	
6	State	SCT	
7	Message	RMC	
8	State	SCT	
9	State	SCT	
10	High-level design	Neither	SCT timed out (10 minutes)
11	High-level design	SCT	
12	State	SCT	
13	State	SCT	
14	State	SCT	
15	Message	RMC	Caught if assuming a transition should only contain 1 message per target
16	State	SCT	
17	*Static*	State reachability	
18	Concurrency	N/A	

Continued on next page

Test	Bug type	Caught by	Notes
19	High-level design	Neither	
20	High-level design	RMC	
21	Message	Neither	
22	Message	RMC	
23	High-level design	RMC	
24	State	STC	
25	High-level design	RMC	
26	State	Neither	
27	State	SCT	
28	Concurrency	N/A	
29	State	STC	
30	High-level design	STC	
31	High-level design	Neither	

Appendix E

Fine-tuning the MESI Protocol

As mentioned in chapter 5, MESI is the only one of the three protocols (as specified by Nagarajan et al. (2020)) considered in this dissertation for which observation 5.3 does not hold. In this appendix, we shall show how it can very easily be adjusted so that the observation holds true.

Recall that observation 5.3 says that the state of the directory should be a function of the state of the caches. For this to not hold true, there then needs to be some cache state for which more than one directory state is possible. In MESI, this arises due to the silent transition from `E` to `M` state in the caches. To see how, consider a very simple system consisting of only one cache and one directory, both initially in explicit state `I`. For simplicity, suppose also that there is no auxiliary state in the system. Now suppose the cache fires a store, sending a `GetM` to the directory. The directory sends back a `GetM_Ack_D`, and now both machines are in `M` state. We have thus learnt that when the cache is in `M`, the directory must be in M[1]. Alternatively, suppose the cache initially fired a load instead. Since the directory is initially in `I`, it would tell the cache it was the first sharer of the line and may in fact hold it exclusively; both machines are now in `E` state. The cache then fires a store, and silently transitions to `M` as per the MESI protocol. Now, we have found an inconsistency in the protocol: we saw that when the cache is in `M` the directory must be in `M` too, yet here the cache is in `M` and the directory is in `E`!

To fix the above problem, all that needs to be done is that when the directory tells the cache to go to `E`, it in fact goes to `M` itself. This is completely equivalent to the previous behavior, because when the directory receives a request in `E` it must act as if the line is being modified anyway, since it cannot know whether the line is clean or dirty. The only adjustment that needs to be made is to allow PutE messages to arrive in the M state of the directory.

The version of MESI which has been adjusted to support observation 5.3 is given in the file `MESI-M.pcc` in our branch of ProtoGen. We have verified it against SWMR and deadlocks using Murphi.

[1]If we had not assumed neither machine has any auxiliary state, we would have had to take the auxiliary state into account too.

Appendix F

Defining the Complexity Measure from Chapter 6

The Complexity Measure (CM) is a primitive attempt at quantifying the complexity of a debugging task, which we used to evaluate Reduced Model Checking and State Consistency Tracking in chapter 6. In this appendix, we define the Complexity Measure in detail.

F.1 Complexity Measure of a General Error Trace

For a general error trace, we define the CM to be two times the number of transitions (note: *not* transactions) in the error trace. This is because at each transition, one needs to consider the messages being sent (or not sent) and the state updates being carried out (or not carried out).

F.2 Complexity Measure of a Reduced Model Checking Error

For Reduced Model Checking, we define the CM based on the type of error thrown by RMC:

- **Tried to send message to the source of a message which is not in scope**: Since this will pinpoint the precise violating message, we take the CM to be 1.

- **Tried to send multiple messages to the same target within a single transition**: Here we take the CM to be the number of surplus messages being sent.

- **Message delivered to machine which could not receive it in its current state**: Since we do not know precisely where things went wrong, but we do know it was in the message flow itself, take the CM to be the number of transitions fired while simulating the NFAs.

- **NFA not accepting**: As above, take the CM to be the number of transitions fired while simulating the NFAs. However, the user will also need to consider the explicit state update in the final transition of the machine, since it may be the case that all messages have in fact been received but the machine simply forgot to update its state. Therefore, add 1 to the CM.
- **Message sent to machine which can never receive such a message**: Take the CM to be 1, since we know precisely which message caused the error.

F.3 Complexity Measure of a State Consistency Error

To define the complexity measure of a state consistency error, we make the simplifying assumption that State Consistency Tracking is run after a Reduced Model Checker with perfect coverage, and so the bug cannot lie in the messages themselves. Thus, we take the complexity measure of a state consistency error to be precisely the number of transitions in the error trace, since only state updates (not messages themselves) need to be considered.