ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications

Citation for published version:
Oswald, N, Nagarajan, V & Sorin, DJ 2018, ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications. in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* Institute of Electrical and Electronics Engineers (IEEE), Los Angeles, California, USA, pp. 247-260, The 45th International Symposium on Computer Architecture, Los Angeles, California, United States, 2/06/18. https://doi.org/10.1109/ISCA.2018.00030

Digital Object Identifier (DOI):
10.1109/ISCA.2018.00030

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)

ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications

Nicolai Oswald
The University of Edinburgh
nicolai.oswald@ed.ac.uk

Vijay Nagarajan
The University of Edinburgh
vijay.nagarajan@ed.ac.uk

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

***Abstract*—Designing directory cache coherence protocols is complicated because coherence transactions are not atomic in modern multicore processors. A coherence transaction comprises multiple messages, and these messages can interleave with other conflicting coherence transactions initiated by other cores. To overcome this architectural challenge, we present ProtoGen, an automated tool for taking the description of a directory protocol with atomic transactions (i.e., no concurrency) and generating the corresponding protocol for a multicore with non-atomic transactions. ProtoGen outputs the finite state machines for the cache and directory controllers, including all of the transient states that are possible with concurrent transactions. We have used ProtoGen to generate complete MSI, MESI, and MOSI protocols given their stable state protocol specifications. We have verified the generated protocols for safety and deadlock freedom using the Murφ model checker. Our generated protocols are identical to or better than manually generated protocols, at times even discovering opportunities to reduce stalling.**

***Keywords*-cache coherence protocols, design automation, hardware synthesis**

I. Introduction

Designing a cache coherence protocol for a multicore processor is a notoriously difficult task. However, when architects are first introduced to coherence protocols, the protocols often seem misleadingly simple, because the protocols appear to have only a small number of clearly defined *stable* states for each cache block (i.e., some subset of the classic MOESI states). As shown in Tables I and II, the specification for an MSI protocol looks fairly intuitive and not overly complicated. For each of a small number of states, the specification describes what happens: (a) when an access (load, store, or replacement) takes place and (b) when an incoming coherence message (e.g., an Invalidate) arrives.

Unfortunately, these textbook protocols with just a handful of stable states are overly simplistic and cannot be applied to typical multicore processors. Specifically, these protocols assume that the transition from one stable state to another stable state is atomic, i.e., the transition happens or appears to happen instantaneously. Only a simplistic system model (e.g., cores connected to a simple atomic bus and a centralized coherence controller) can provide this atomicity, yet high-performance multicores employ multi-hop interconnection networks and distributed directory protocols.

Typical multicore directory protocols are complicated because of their non-atomic system model. When a cache performs a coherence transaction to change a block from one stable state to another, the transaction typically involves multiple steps (issuing a request, waiting for a response, etc.) that could potentially interleave with other conflicting coherence transactions initiated by other caches. Designing protocols to correctly handle this concurrency is challenging.

The protocol complexity introduced by concurrency is revealed in the large number of possible *transient* coherence states, in addition to the handful of stable states. At each step in a coherence transaction, the cache usually changes the state of the block to a different transient state that reflects that step in the transaction. For example, consider a cache that has a block in state I(nvalid) and issues a request for state S(hared). In the interim—between issuing the request and when a response arrives to complete the transition from I to S—the cache block will be in an at least one transient state. Moreover, in addition to the transient states between a current state and a requested state, there are often transient states that arise due to coherence requests from other caches that arrive in this window of time. Even a relatively simple MSI directory protocol has eighteen transient states, as shown in Table VI.

In typical coherence protocols with dozens of states, it is easy for architects to make mistakes. An architect can forget about a possible coherence message that can arrive for a block in a certain state. An architect can introduce a bug in how an incoming message is handled for a block in a certain state. An architect can fail to think of a possible situation and end up with too few states. Subtle bugs in the coherence protocols can seriously affect end users; for example, a bug in the CCI-400 cache coherent interface caused the Samsung Galaxy S4 to ship with coherence disabled, with serious performance/power implications [1]

In addition to these safety problems, a protocol can also fail to achieve its maximum performance because an architect can conservatively restrict concurrency in overly complicated situations. Rather than reason about how to handle a certain message that arrives for a block in a certain state, it can be tempting to simply stall that request until the block is in a stable state.

Table I
Specification of Cache in Atomic MSI Protocol

	Load	Store	Replacement	Forw. GetS	Forw. GetM	Invalidation
I	send GetS to Dir, receive Data / S	send GetM to Dir, receive DataNoAck or Data and Acks / M				
S	hit	send GetM to Dir, receive DataNoAck or Data and Acks / M	send PutS to Dir, receive Put-Ack / I			send Ack to requestor / I
M	hit	hit	send PutM to Dir, receive Put-Ack / I	send Data to requestor and Dir / S	send Data to requestor / I	

Table II
Specification of Directory in Atomic MSI Protocol

	GetS	GetM	PutS	PutM
I	send Data to requestor, add requestor to Sharers / S	send Data to requestor, set Owner=requestor / M		
S	send Data to requestor, add requestor to Sharers	send Data to requestor, send Invalidations to Sharers, set Owner=requestor, clear Sharers / M	send Put-Ack to requestor, remove requestor from Sharers	
M	forward GetS to Owner, receive Data from Owner, add requestor and Owner to Sharers / S	forward GetM to Owner, set Owner=requestor		send Put-Ack to requestor / I

Thus we have a situation in which we can easily understand protocol specifications with just the stable states, yet current multicore system models require that we have many transient states and the resultant complexity.

In this paper, we present ProtoGen, an automated tool for taking a stable state protocol (SSP) specification of a directory coherence protocol and generating a directory protocol with those same stable states and all of the transient states needed to maximize protocol concurrency, while preserving safety (correctness) and preventing deadlocks. ProtoGen accepts the SSP specification in a domain-specific language and generates finite state machines for the cache controller and directory controller.

ProtoGen overcomes the key challenge of protocol generation—creating cache and directory controllers that correctly handle incoming coherence messages when transactions are racing—by leveraging the insight that, in a directory based coherence protocol, racing transactions are serialized at the directory. By assigning a unique name to every directory-forwarded request that can arrive at a stable state in a cache, ProtoGen makes it possible for the directory to convey this serialization order to the caches. With the caches and the directory achieving consensus on the order of racing transactions, ProtoGen is able to generate highly-concurrent and non-blocking controller actions that are consistent with this order.

ProtoGen has similarities to prior work on high-level synthesis of hardware [2], [3], [4] but, to the best of our knowledge, it is the first tool for generating concurrent cache coherence protocols given SSP specifications. In this paper, we show that it correctly infers all of the transient states and produces verifiably correct high-performance directory protocols for multicore processors.

In summary, the contributions of this paper are as follows.

- We present ProtoGen[1], an automated tool for taking an SSP specification of a directory coherence protocol and generating a complete directory protocol with those same stable states and all of the transient states needed to maximize protocol concurrency, while preserving safety and preventing deadlocks.
- We have used ProtoGen to generate high-performance MSI, MESI, and MOSI protocols given SSP specifications. We have verified the generated protocols for safety and deadlock freedom using the Murφ model checker [5]. Our generated protocols are identical to or better than manually generated high-performance protocols [6], at times even discovering opportunities to reduce stalling.
- We further demonstrate ProtoGen's versatility by using it to generate a complete TSO-CC protocol [7] given its SSP specification. TSO-CC is an unconventional protocol that is designed specifically to satisfy the TSO memory consistency model.

II. Background and Related Work

At its heart, ProtoGen is a method for refining an atomic specification into a non-atomic implementation. Therefore, we first discuss prior approaches for realizing the same. We then discuss how ProtoGen relates to program synthesis based techniques for synthesizing a complete coherence protocol given an incomplete one with *holes*. Finally, we discuss work that has proposed new coherence protocols for mitigating the design and verification effort involved.

[1] https://github.com/icsa-caps/ProtoGen

A. *From Atomic Specification to Implementation*

Deriving a high-performance concurrent implementation from an atomic specification is a topic that has received significant attention and finds application in a number of areas including databases (high-performance transactions), programming languages (software and hardware transactional memory) and hardware synthesis (e.g., Bluespec [8]).

An atomic specification is essentially a sequence of steps (a transaction) that transforms some state (e.g., a set of objects) as if it were the only agent modifying the state. Broadly speaking, there are two approaches to get to a concurrent implementation from an atomic specification: a blocking approach (using locks) and a non-blocking approach (without locks).

Non-blocking techniques [9], because they eschew locking, typically allow for a higher degree of concurrency. Such techniques can further be divided into two categories. The first category (often referred to as lock-free programming) uses complex algorithms that allow for multiple (conflicting) transactions to overlap with each other while relying only on atomic single-word primitives provided by the hardware (e.g., compare-and-swap) for providing the illusion of atomicity. In contrast, optimistic concurrency control allows for conflicting transactions to overlap speculatively, relying on the underlying software or hardware runtime to detect and recover from conflicts. We now focus on techniques that use atomic specifications for hardware synthesis in general and coherence protocols in particular.

General Hardware Synthesis. Atomic specifications have been used for hardware synthesis [10], most notably by the Bluespec [11], [8] line of work. The essential idea is to specify the behavior of a hardware module as *guarded atomic actions*: *rules* that get triggered upon a rule condition, and atomically update multiple pieces of state (e.g., registers, RAMs). Their compiler would then produce a concurrent hardware implementation in RTL (Verilog or VHDL). To this end, the compiler first analyzes the rules in order to identify *rule conflicts* (i.e., rules that read and write to the same state) and then synthesizes a hardware scheduler that ensures that conflicting rules are not scheduled concurrently in the same cycle. Thus, the compiler typically produces a blocking implementation. Furthermore, for practical reasons the compiler only generates actions that complete in a single cycle. It is worth noting that there has been some work [12], [13] that has attempted to lift these restrictions, although it is unclear whether these have been integrated into the Bluespec compiler.

Blocking Protocol Synthesis. An SSP protocol can potentially be specified in Bluespec by expressing every SSP transition as a Bluespec rule. While the Bluespec compiler will synthesize a concurrent implementation (by ensuring that conflicting rules are not scheduled concurrently), the protocol will be blocking in nature. Specifically, any two racing coherence transactions (to the same cache line) would be deemed conflicting and therefore completely serialized, which has the potential to limit performance significantly. In fact, Atomic Coherence [14] employs a very similar strategy. It argues for atomic protocols (much like our SSP specifications), and proposes a mutex based approach (in hardware) to guarantee atomicity. However, performance was achieved by leveraging optical interconnects for low-latency mutexes.

Non-blocking Protocol Synthesis. Dave et al. [2] presented a case study that showed that realistic non-blocking coherence protocols can be expressed naturally in Bluespec; they also showed that the synthesized coherence controllers met the required timing constraint. However, their case study did not take an SSP protocol as input; rather it took a correct and complete non-blocking MSI protocol (with all of the transient states and concurrency) as input to obtain RTL (Verilog) output. In other words, their generated protocol had the same degree of concurrency as the specification.

ProtoGen. Unlike Bluespec, ProtoGen is not a general synthesis tool and is limited to synthesizing directory based cache coherence protocols. But, by exploiting domain knowledge about how directory protocols work, ProtoGen is the first tool that can produce a non-blocking protocol given an SSP.

B. *Coherence Protocols via Program Synthesis*

TRANSIT [3], [15] allows a protocol designer to specify only the skeleton of a coherence protocol; the missing state transitions or *holes* are synthesized using an iterative usage model as follows. The designer provides information that describes each hole via concrete or symbolic (i.e., *concolic*) snippets. TRANSIT then synthesizes each hole independently from the snippets, thereby sidestepping the state explosion of synthesizing all holes together. Whether or not the individually synthesized holes combine to form a correct protocol is then verified with an external model checker. If the resulting protocol turns out to be incorrect, the whole process has to be repeated; indeed, the user has to integrate counter-examples from the model checker into the skeleton to better guide the synthesizer in the next iteration.

VerC3 [4] strives to improve automation in the synthesis process, by relying only on protocol properties without the user providing example traces or other hints to the synthesizer. They employ an explicit state model checker for synthesis and make use of a candidate pruning optimization for significantly reducing the search space. Despite this, the number of holes they can synthesize is limited due to state explosion; given an SSP for an MSI protocol, they are able to synthesize only 12 out of a possible 35 holes.

In contrast to the above approaches that *search* for the missing holes, ProtoGen views the problem as refinement from an atomic specification to an equivalent concurrent implementation. By exploiting domain knowledge about directory protocols, ProtoGen can produce a high-performance non-blocking protocol. Thus, it avoids the state explosion problem experienced by VerC3 and the user intervention in TRANSIT.

C. *Complexity-aware Coherence Protocols*

There have been a number of proposals that have devised new ways of achieving coherence that mitigate the design and verification costs of typical coherence protocols. DeNovo [16] and VIPS [17] are examples of protocols that reduce coherence protocol complexity by exploiting data-race-free models to minimize transient states. Fractal coherence [18] proposes a methodology for designing coherence protocols that makes them amenable to existing formal verification tools. Orthogonal to the above work, ProtoGen is not a new protocol—rather it is a method for generating a complete non-blocking protocol given an SSP specification.

III. Intuition for ProtoGen

Before delving into the low-level details of ProtoGen, we first present a high-level explanation of how it works. As part of this explanation, we introduce some background material. Where possible, we adopt the terminology and notation used in Sorin et al.'s primer on consistency and coherence [6].

A. *System Model and Terminology*

ProtoGen is designed for multicore processors with flat directory cache coherence protocols. Thus far, we assume that the protocols use a subset of the well-known MOESIF states. Furthermore, we assume typical coherence requests to either increase permissions (with a "Get" or "Upgrade") or decrease permissions (with a "Put"). These protocol assumptions are likely more conservative than necessary, but we have not yet explored protocols with other states or request types.

Each core can have one or more levels of private cache, and there is a last-level cache (LLC) that is shared by all cores. For simplicity, we describe systems with one level of private cache, but more levels are possible. The directory state is colocated with the LLC. We make no assumptions about the interconnection network, regarding either topology or whether point-to-point ordering is enforced.

For clarity, we now define several terms that we use throughout this paper. We consider every cache block and every directory entry to have a *state* that consists of its *coherence permission state* and possible *auxiliary state*. Example coherence permission states are S(hared) and M(odified), and they reflect a cache's ability to access a block or a directory's knowledge about a block. Auxiliary state is any state that does not describe permissions, but is often necessary to correctly transition between two coherence permission states. A directory entry's auxiliary state might include the ID of the current owner of the block and the set of caches currently sharing the block (the sharer list). A cache block's auxiliary state might include a counter used to keep track of incoming acknowledgments.

If a core wants to perform an *access* (load, store, or replacement) that cannot be satisfied by its cache, its cache initiates a coherence *transaction* to obtain the desired cache block in the appropriate coherence state or evict it. A transaction consists of:

- an initial *request* message such as GetShared (GetS), GetModified (GetM), or PutModified (PutM);
- zero or more *forwarded* requests—such as Forwarded-GetM or Invalidation messages—that are sent by the directory in response to incoming requests;
- one or more data *responses* or *acknowledgments*. Both the directory and the cache controller may respond to incoming coherence messages with data and acknowledgment messages. For example, if a directory with a block in state I(nvalid) receives a GetS request, it will respond with Data; and
- state *transitions* in response to incoming coherence messages. A cache may change its block state and a directory may change the state of its entry; these state changes can include the coherence permissions and/or the auxiliary state. For example, if a directory receives a GetS from a cache for a block in state I, it will change the directory entry's coherence permission state and its auxiliary state for tracking sharers.

In our examples, we often consider a transaction from the point of view of a given cache that initiates the transaction with a request. We refer to that transaction as the cache's "own" transaction. If that cache receives a message that is part of a transaction initiated by another cache, we refer to that as an "other" transaction. Similarly we refer to coherence messages as "own" (e.g., own GetM) and "other" (e.g., other PutS).

Lastly, we consider each transaction to start or end a *coherence epoch*, a window of time during which a cache has coherence permission to a block. For example, a cache issues a GetS request to start its own transaction that, when complete, will begin a Shared epoch at that cache. This epoch will end either when: (a) the cache completes another transaction to change its permissions, or (b) another cache performs a transaction that affects the cache's permissions.

B. *Definition of Coherence*

ProtoGen generates coherence protocols that adhere to certain safety properties. We use the common definition of coherence, which consists of two invariants. First, for any memory location at any given time, there is either a single writer or zero or more readers. This invariant is often referred to as SWMR, and it means we can divide a block's lifetime into epochs, during each of which there is either a single writer or zero or more (multiple) readers. Second, the value of a location at the start of an epoch is the same as the value of the location at the end of its last write epoch. This invariant is often referred to as the data-value invariant.

Intuitively, these epochs can be in physical time, but it is also correct for them to be in logical time, in which the two invariants combine into one single invariant, i.e., sequential consistency for every memory location [19]. A discussion of logical time correctness is beyond the scope of this paper, but numerous protocols have been developed that rely upon it [20], [21], [22]; the benefit of per-location sequential consistency is that it can enable greater concurrency.

C. Big Picture

ProtoGen starts with a SSP and creates a complete protocol with transient states—generating both coherence permission and auxiliary state—without requiring an atomic system model that can guarantee physically atomic transactions. ProtoGen requires that the SSP descriptions for the cache and directory are correct and complete for an atomic system model; for example, the SSP must correctly enforce SWMR.

In this discussion, we focus on the generation of transient coherence permission states, because the issues involved in generating transient auxiliary state are either the same or far simpler. Intuitively, auxiliary state is either intimately tied to the coherence state (e.g., a directory entry in S keeps auxiliary state to track the sharers) or is "bookkeeping" state that is uninvolved in races (e.g., a counter for a cache block to track how many acknowledgments it has received).

To understand how ProtoGen works, first consider a simplistic protocol without physically atomic transactions but with no concurrency either and hence logically atomic transactions (i.e., for any given block of memory, only one coherence transaction can be active at a time). Generating transient state specifications for such a protocol is simple; each transient state just corresponds to a step in the transaction. For example, a cache issuing a GetS request to transition from I to S has a single transient state, which we will call IS, that reflects it has issued the request but has not yet received the response; once it receives a Data response, it will transition to S.

The challenge for ProtoGen is handling multiple concurrent transactions for a given block. What happens when a cache in the middle of a transaction receives a forwarded request belonging to a concurrent transaction to the same block? In our above example, what happens when the cache that issued the GetS receives an Invalidation from another cache (via the directory) while in state IS? In directory protocols, the directory is the serialization point, so the key is for the cache to be able to deduce the ordering of transactions at the directory. In our example, the cache must deduce whether its own GetS was ordered at the directory before or after the other cache's transaction that led to the incoming Invalidation. Here, the cache can infer that its own GetS was ordered at the directory first; otherwise there is no reason for the directory to have forwarded the Invalidation. Thus, by simply looking at the incoming forwarded request, ProtoGen is able to deduce ordering.

But the SSP could have been written such that the same forwarded message could arrive in two stable states. To deal with this, ProtoGen preprocesses the input SSP to ensure that a given forwarded request can arrive at exactly one stable state (if the input SSP uses the same name for two forwarded request messages, ProtoGen creates a new name for one of them). This invariant allows for caches to reliably deduce the order in which transactions are serialized at the directory by simply looking at incoming forwarded requests. ProtoGen uses this ordering information to generate the cache and directory state machines as described in detail later.

Intuitively, ProtoGen creates transient states for the caches and directory so that these finite state machines always respect the ordering of transactions at the directory. By doing so, ProtoGen creates protocols that are guaranteed to enforce coherence.

IV. Using ProtoGen

In this section, we discuss the input, output, and limitations of ProtoGen.

A. Input

The primary input to ProtoGen is a high-level specification of a stable state protocol (SSP) described in our domain specific language (DSL). Our DSL is similar in spirit to other previously proposed ones for coherence protocols, including Teapot [23] and SLICC [24]. The specification describes the protocol with respect to a single cache block, because the behavior of all blocks is identical, and it essentially contains the information in Tables I and II, including:

- a list of stable coherence permission states, often a subset of the MOESIF states;
- the stable auxiliary state at each cache block and each directory entry;
- a list of accesses (loads, stores, replacements) and the requests they trigger;
- a list of possible coherence messages (requests, forwarded requests, responses, and acknowledgments) that can arrive in each stable state; and
- the transitions from one stable state to another stable state—including both coherence permission and auxiliary state—that occur as a result of incoming coherence messages.

The DSL enables the designer to define the structure of any machine (e.g. caches and directories) as well as its architectural behavioral specifications. To this end, the DSL supports standard data types like bool, int, and set, but there are also protocol-specific data types. In particular, the DSL provides a special type called Data (to represent actual data in a block of memory). Every machine has two predefined internal variables: State (denoting the state of the machine) and ID (the ID of the machine).

Listing 1 shows a snippet of code that uses many of these data types in defining the cache structure (lines 2 through 7). In line 3, we have initialized State to I. In lines 5 and 6, we have defined and initialized two auxiliary states: $acksReceived$ and $acksExpected$.

```
1  // Machine definition
2  Cache {
3      State = I;          // Cache initial state
4      Data block;
5      int[0..NumCaches] acksReceived = 0;
6      int[0..NumCaches] acksExpected = 0;
7  } set[NumCaches] cache;
8  ...
9
10 Architecture cache {
11     ...
12
13
14     // S to M transition
15     Process(S, store){
16         msg = Request(GetM, ID, dir.ID);
17         reqNet.send(msg);
18         acksReceived = 0;
19
20         await{
21             when GetM_NoAck:
22                 State = M;
23                 break;
24
25             when GetM_Ack:
26                 acksExpected = GetM_Ack.acksExpected;
27
28                 if acksExpected == acksReceived{
29                     State = M;
30                     break;
31                 }
32
33                 await{
34                     when Inv_Ack:
35                         acksReceived = acksReceived + 1;
36
37                         if acksExpected == acksReceived{
38                             State = M;
39                             break;
40                         }
41                 }
42
43             when Inv_Ack:
44                 acksReceived = acksReceived + 1;
45         }
46     }
47     ...
48 }
```

Listing 1. Snippets from ProtoGen DSL

Our DSL also allows us to specify the behavior of the caches and directories in response to internal accesses and incoming coherence messages. Most of these transactions consist of a request and a response, and specifying these transactions is straightforward. However, there are two specification issues that are worth discussing.

First, some requests can lead to multiple possible transaction routes depending on the state of the block in other caches; our DSL allows the user to specify this. Consider the S to M transition (lines 14 through 45), in which a cache—upon receiving a store to a block in shared state—requests Modified access to the block. If the block is exclusively held by the requestor and not cached elsewhere, the requestor simply receives a single response from the directory to complete its transaction (lines 21 through 23). If, however, the block is held by one or more other caches, the requestor must wait for both the response from the directory (containing a count of the sharers) and the acknowledgments from all caches that had the block in state S (lines 25 through 44).

This example also leads us to the second issue: some transactions require the initiating cache to receive multiple *types* of messages to complete the transaction. Our DSL allows the user to specify that multiple messages must arrive. Coming back to our example, the two relevant messages are $GetM\_Ack$ (the message from the directory with the count), and Inv_Ack (invalidation acknowledgment). When the former arrives (line 25), our DSL allows us to set the auxiliary state $acksExpected$; when the latter arrives (line 34), $acksReceived$ is incremented; when they are are found to be equal, the transaction completes. [2]

Configuration parameters. ProtoGen also has inputs that control the nature of the protocols that it generates. One parameter controls whether the generated protocol is *stalling* or *non-stalling*. With the former, cache and directory controllers stall when they receive potentially racing requests, at the cost of performance (while still preventing deadlocks). With the latter, the generated protocol avoids stalling whenever possible at the expense of an increase in the number of transient states.

Another ProtoGen parameter controls whether the generated protocol allows for loads or stores to access a block in a transient state (e.g., a load to a block in a transient state between S and M), and this input affects the coherence invariant that the generated protocol guarantees. Allowing accesses to cache blocks in transient states can preclude a protocol from enforcing SWMR in physical time but is compatible with enforcing per-location sequential consistency.

B. Output

ProtoGen produces fine state machines (FSMs) for the caches and the directory including the transient states (containing information similar to Table VI). These FSMs are expressed in the same DSL and can be translated to any other format for specifying FSMs. Thus far, we have implemented a backend to the language of the Murφ model checker [5] and translation to other outputs like SLICC [24] or Verilog is future work.

C. Limitations

First, ProtoGen requires a correctly specified SSP as its input; it cannot automatically "correct" bugs in the SSP. Second, ProtoGen cannot generate new protocol actions not explicitly specified in the SSP. For example, it cannot infer how atomic read-modify-writes must be implemented without it being specified in the SSP; likewise, it cannot automatically

[2] Due to races, an Inv_Ack can actually arrive before the $GetM\_Ack$, which is the situation handled by lines 43-44.

deduce the protocol for an unordered network given an SSP for an ordered network. Third, ProtoGen does not specify how protocol actions must interact with the interconnect; it requires the user to manually define virtual channels and assign messages to channels. Finally, ProtoGen is limited to directory based protocols.

V. PROTOGEN

Given an SSP, ProtoGen generates a highly concurrent directory protocol including the transient states. This process is explained step by step in this section, and we use a running example of an MSI protocol to illustrate each step. We first explain this process for generating the cache controller; at the end of this section, we discuss the minor differences involved in the process of generating the directory controller.

Unless otherwise noted, we focus on the coherence permission state, because incorporating the stable auxiliary state is usually trivial. We only discuss the auxiliary state when ProtoGen has to generate transient auxiliary state (e.g., state for a cache block that records to which other cache to send the data when its own transaction completes).

A. Preprocessing the SSP

Before generating any transient states, ProtoGen first preprocesses the SSP specification to ensure the invariant that a given forwarded request can arrive at exactly one stable state. If, in an input SSP specification, the same forwarded request can arrive in two stable states, ProtoGen creates a new name for one of the forwarded requests.

For some directory protocols, architects might find it natural to specify their SSPs in a manner that already satisfies the invariant. For example consider the SSP of the MSI protocol specified in Table 1. As we can see, each of the three forwarded requests—Fwd_GetM, Fwd_GetS and Invalidate—arrive at exactly one stable state: M, M, and S respectively.

Table III
MOSI SSP: BEFORE PREPROCESSING

	Fwd_GetS
M	send Data to requestor / O
O	send Data to requestor

Table IV
MOSI SSP: AFTER PREPROCESSING

	Fwd_GetS	O_Fwd_GetS
M	send Data to requestor / O	
O		send Data to requestor

On the other hand, consider a MOSI protocol. Architects might find it natural to specify its SSP such that a Fwd_GetS can arrive at both the M and O states (the relevant snippet of the SSP is shown in Table III). In such a case, ProtoGen would rename one of the messages as shown in Table IV. If the directory receives a GetS and finds the block in O state, the directory would forward the new O_Fwd_GetS message.

To see *why* this renaming is necessary for ProtoGen, let us consider the following scenario. Consider a cache C_0 that holds a block in O state and wants to write to the block. Accordingly, it would send a GetM request to the directory and wait for a response. In the meantime, say there is a concurrent transaction to the same block: another cache C_1 wanting to read the same block issues a GetS to the directory. The key point to note here is that the directory will forward the GetS to C_0 irrespective of the order in which the two transactions serialize at the directory. But for ProtoGen to work, C_0 needs to somehow discover the order in which the racing transactions have serialized at the directory. It is for this very reason that ProtoGen performs the renaming: if C_0 were to receive a Fwd_GetS message it can now infer that its own GetM request must have “won the race”; if C_0 were to receive a O_Fwd_GetS on the other hand, it can infer that the other GetS request must have been serialized before its own GetM.

B. Step 1: Generate Initial State Sets

The key challenge that ProtoGen addresses is to generate cache controllers that correctly respond to incoming forwarded requests for a block in a transient state. But for this, ProtoGen must first know what forwarded requests can *potentially* arrive in a transient state.

It is worth noting here that transient states are local to a cache and not visible to the directory. The directory always sees any block in a cache as being in a stable state at all times; it forwards requests to a cache based on the state of the cache block as it sees it. Therefore, the set of forwarded requests that can arrive at a transient state is determined by the set of possible stable states in which the block can be seen at the directory (while the cache block is in the transient state).

ProtoGen keeps track of this information with a data structure called a State Set. ProtoGen creates one State Set for each stable state, and initially each State Set contains just its stable state. For an MSI protocol, the State Sets are initially $\{\mathrm{I}\}$, $\{\mathrm{S}\}$, and $\{\mathrm{M}\}$, and we refer to them as the $\boldsymbol{I}$, $\boldsymbol{S}$, and $\boldsymbol{M}$ State Sets, respectively. (We use **bold** to distinguish a State Set from a stable state.) As ProtoGen generates new transient states, it adds them to one or more State Sets, as described below.

C. Step 2: Add Transient States in Absence of Concurrency

ProtoGen next adds the transient states required for non-atomic protocols in the absence of concurrency. If a transaction from stable state S_i to stable state S_j involves issuing a request and waiting for a single response, ProtoGen would add a single transient state that reflects the situation in which the request has been issued but the response has not yet been received. If the transaction involves multiple responses, ProtoGen would add a transient state for each response.

In our MSI protocol, a cache can initiate one of five transactions types, $I \rightarrow S$, $I \rightarrow M$, $S \rightarrow M$, $S \rightarrow I$, and $M \rightarrow I$. The transaction from I to S would lead to the creation of a transient state that we call IS, that reflects the situation when the cache has issued the GetS request to the directory and is awaiting a data response. When the data is received from the directory, the cache will transition from IS to S.

Recall that the transaction from I (or S) to M could happen via two routes depending on the state of the block at the directory: one route with a single data response (if the block is in I or M at the directory) and the second route with multiple responses. To account for this, ProtoGen creates transient states as shown in Table V. The transaction from I to M would first lead to the creation of a transient state that we call IM^{AD} (potentially waiting for data as well as acknowledgments). If the response received consists of only data, the cache block will directly transition to M state. If the response includes both data and a count of acknowledgments, then the block will transition to another transient state called IM^{A} (waiting for acknowledgments). When the last of the acknowledgments is received, the block will transition to M.

Table V
ADDING TRANSIENT STATES (NO CONCURRENCY)

	Store	DataNoAcks	Data + #Acks	Last Ack
I	send GetM to Dir / IM^{AD}			
IM^{AD}		M	IM^{A}	
IM^{A}				M

When ProtoGen creates a new transient state for a transaction from stable state S_i to stable state S_j, it adds it to the State Set for both S_i and S_j. This "duality" is because the transient state can behave like either S_i or S_j, depending on what message arrives (i.e., whether the message is one that could arrive in S_i or one that could arrive in S_j). In our example, state IS thus belongs to the State Set $\boldsymbol{I}$ and the State Set $\boldsymbol{S}$. At the end of this step, the State Sets for our MSI protocol are as follows:
$\boldsymbol{I}$=$\{I, IS, IM^{\text{AD}}, SI, MI\}$,
$\boldsymbol{S}$=$\{S, IS, SM^{\text{AD}}, SI\}$, and
$\boldsymbol{M}$=$\{M, IM^{\text{AD}}, IM^{\text{A}}, SM^{\text{AD}}, SM^{\text{A}}, MI\}$.

D. Step 3: Accommodating Concurrency

We now explain how ProtoGen accommodates concurrency by describing the generated cache controller behavior for when a forwarded request arrives in one of the transient states produced in Step 2. In this section, we explain how this is done for any given transient state. Later in Section V-G, we present a global picture of how ProtoGen does this for *all* transient states.

A cache that has a block in any state in State Set $\boldsymbol{S_i}$, can receive any forwarded request that the SSP specifies is possible to arrive in stable state S_i. If the message arrives in that stable state, the block's state will immediately change to a stable state (or perhaps remain in S_i) as specified in the SSP. The more challenging scenario is if the forwarded request arrives in a transient state (belonging to State Set $\boldsymbol{S_i}$). In our example, an Invalidation arriving in stable state S is easy to handle; the block immediately transitions to stable state I. But what if the Invalidation arrives in transient state IS? Because ProtoGen's preprocessing step ensures that any type of forwarded request can arrive only in a single stable state, ProtoGen can generate the new transient states without confusion.

Consider a transient state that is part of cache C_0's transaction T_{own} (triggered by an access A_{own}) from stable state S_i to stable state S_j. This transient state, which we call S_{ij}, is part of State Sets $\boldsymbol{S_i}$ and $\boldsymbol{S_j}$, and thus any forwarded request that can arrive at C_0 in state S_i or S_j can also arrive in S_{ij}. ProtoGen considers every one of these possible forwarded messages and how a cache in S_{ij} would respond to it. As explained in Section III, the key is knowing in which stable state that forwarded request can arrive, and thus inferring the transaction ordering at the directory.

We now consider the two possible scenarios in which a forwarded request arrives at C_0 while it is in a transient state: either the forwarded request was ordered earlier or later than C_0's transaction T_{own}.

1) Case 1: Other Transaction Ordered Earlier: If the arriving forwarded request (R_{other}) is one that is associated with state S_i, C_0 can infer that the directory must have seen the other transaction before its own request (R_{own}), i.e., $T_{other} \rightarrow T_{own}$ at the directory. Moreover, C_0 can infer that when T_{other} arrived at the directory, the directory must have seen C_0 in state S_i. (At this instant C_0 is unable to infer whether or not its own request has reached the directory, but this is not needed.)

Let us assume that in the SSP the forwarded request R_{other} causes C_0 to transition from S_i to S_l. Upon receiving R_{other}, C_0 must: (a) respond to this request immediately; (b) transition to a transient state and logically restart its own transaction T_{own} as if starting from the stable state S_l. Let us now discuss the two issues in more detail.

Responding to forwarded request immediately. Once C_0 infers that R_{other} is part of an earlier transaction than T_{own}, it is critical that C_0 respond immediately to R_{other}. In particular C_0 must not wait for a response for its own request, as this could potentially lead to a deadlock.

To see why, consider two caches C_0 and C_1 both wanting to transition from S to M state, so both caches issue a GetM to the directory; let us refer to the two racing transactions as T_0 and T_1 respectively. Say the GetM from C_1 "won the race" and reached the directory first—i.e., $T_1 \rightarrow T_0$—but the response from the directory was delayed and instead C_1 received the forwarded GetM request that is part of T_0 first. C_1, upon seeing the forwarded GetM can infer that its own request won the race (otherwise the directory would not have forwarded the GetM), and so it can choose to stall the incoming forwarded GetM request (more about stalling in Section V-D2). However, if C_0 also chose to delay the incoming Invalidate (that is part of T_1), this will lead to a circular dependency between T_0 and T_1 and hence a deadlock. This explains why C_0 must respond to the Invalidate immediately once it knows that the Invalidate is part of the earlier transaction.

Transitioning to a suitable transient state. Once C_0 responds to the incoming forwarded request, what state must it transition to? Logically, C_0 must appear as if it first transitioned to S_l and then performed the access A_{own}. But the problem here is that C_0 had already sent a request R_{own} (for access A_{own}) to the directory when in stable state S_i. Technically, the earlier request must be rescinded and a fresh request must be sent. However, for most directory protocols the same memory access in two stable states triggers the same request to the directory. In such a case—i.e., if the access A_{own} triggers the same request R_{own} from S_l also—there is no need for C_0 to rescind the earlier request and send a new request from S_l. It can simply move to a transient state that logically corresponds to the situation in which it has issued R_{own} from S_l and is waiting for a response. If such a transient state does not already exist, ProtoGen creates it. Say in the SSP the request R_{own} causes a transition from S_l to S_m. The transient state to enter would be S_{lm} between these two stable states, and it would have been identified in Step 2 (Section V-C).

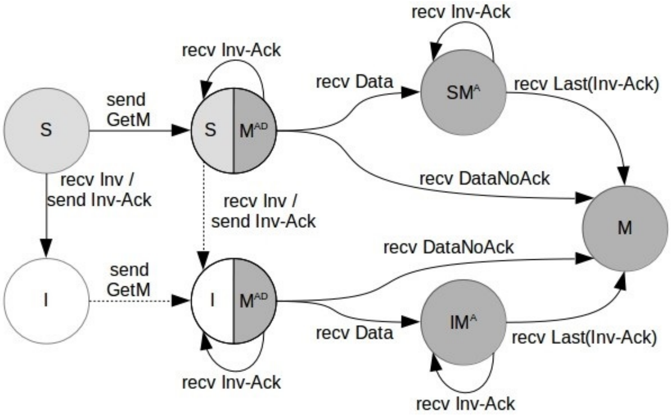

Figure 1. Cache S to M Transaction with $T_{other} \rightarrow T_{own}$. The shade of each state denotes the State Set(s) it belongs to. E.g., IM^{AD} is part of I and M State Sets

In our MSI protocol, consider the transient state SM^{AD} shown in Figure 1, which denotes that the cache has issued a GetM request to transition from S to M but has not yet received a response. If an Invalidation arrives, which is only possible in stable state S, the cache infers that the other transaction (involving the Invalidate) was ordered before its own transaction. ProtoGen looks at the SSP and discovers that an Invalidate received in state S would send the block to state I. Logically, ProtoGen must make the cache appear as if it first went to state I and then performed a store. But recall that the cache had already issued a GetM request to the directory, in response to a store in state S. Fortunately, the SSP reveals that a store in state I results in sending the same request, i.e., GetM. Thus, there is no need to rescind the earlier GetM. Further, ProtoGen would check the complete protocol generated until now to look for a transient state that denotes the situation in which the cache has issued a GetM in I and is waiting for a response; it is able to find IM^{AD}, and so ProtoGen transitions from SM^{AD} to IM^{AD}.

However, in some situations the same access could lead to two different requests when in two different stable states. Consider an MSI directory protocol that uses *Upgrade* requests. An Upgrade is a special type of request for transitioning from the S to M state. The difference between an Upgrade and a GetM is that the former does not need the data from the directory whereas the latter requires it. Consider two caches C_0 and C_1, both wanting to transition from S to M state, so both caches issue an Upgrade to the directory; let us refer to the two racing transactions as T_0 and T_1 respectively. Suppose C_1 won the race and reached the directory first. Logically, C_0 must now issue a GetM as the data it holds in its cache is invalid; it must obtain the new data written by C_1. Thus, this is an example where the same access (store) can lead to two different requests: Upgrade (if block in S) or GetM (if block in I). ProtoGen deals with this issue as follows. When the directory receives an Upgrade from C_0, it infers what happened because it knows that an Upgrade request cannot possibly arrive in state I. The directory reinterprets the Upgrade as a GetM, the request triggered by the same access (store) in state I.

2) Case 2: Other Transaction Ordered After: Once again, consider a cache C_0 that has issued a request to transition from S_i to S_j and is in a transient state that reflects that it has issued the request but has not received the response. If an arriving forwarded request (R_{other}) is one that is associated with state S_j, C_0 can infer that the directory must have seen C_0's own request (R_{own}) before the other transaction (i.e., $T_{own} \rightarrow T_{other}$ at the directory). Moreover, C_0 can infer that when T_{other} arrived at the directory, the directory must have seen C_0 in state S_j, which is why it forwarded R_{other} to C_0 in the first place.

Let us assume that in the SSP the forwarded request R_{other} causes C_0 to transition from S_j to S_k. Upon receiving R_{other}, C_0 must logically transition to S_k, but C_0 is unable to enter the stable state S_k yet because it is still waiting for a response to its own earlier request R_{own}. C_0 must honor the ordering $T_{own} \rightarrow T_{other}$, and there are three approaches to doing so.

Stalling. The most straightforward way to honor this ordering is by *stalling* C_0 and not having it respond to the forwarded request R_{other} until a response to its own request R_{own} has been received. Because the later transaction is the one that is stalled, there is no risk of a deadlock. However, stalling degrades performance in two ways. First, stalling will delay the start of the coherence permission epoch that R_{other} seeks to initiate. Second, stalling the controller will also block incoming coherence messages for other cache blocks.

Immediate Transition, Deferred Responses. ProtoGen can generate cache controllers that achieve greater performance by *not* stalling when the forwarded request R_{other} arrives. The key observation is that C_0 can process R_{other}—and avoid having it block its incoming queue—and any subsequent forwarded requests that arrive for the same block, knowing they are all ordered after T_{own}. C_0 enters a new transient state S_{new}. Because R_{other} causes a transition from S_j to S_k, the new transient state S_{new} is inserted into the State Set of $\boldsymbol{S_k}$. If the arrival of R_{other} in S_j would cause the sending of one or more responses, then C_0 *defers* the sending of these responses until it has completed its own transaction.

In a similar vein, if any subsequent forwarded request for the same block (say R_{other2}) arrives at C_0 in S_{new}, C_0 can infer that R_{other2} is part of a transaction that is ordered after T_{other}.[3] Therefore, upon receiving a forwarded request while in S_{new}, C_0 behaves analogously to how it behaved when R_{other} arrived. It transitions to another transient state S_{new2} and, if the arrival of R_{other2} in S_k would cause the sending of one or more responses, then C_0 also defers sending those responses.

A cache that processes incoming forwarded requests instead of stalling may need transient auxiliary state to remember where to send the responses it defers. ProtoGen generates this transient auxiliary state when it generates cache controllers. It may appear that this state is unbounded, because each subsequent forwarded request could require some amount of state. In most directory protocols, however, the number of forwarded requests that a cache can receive is limited to three or fewer, before the cache block will reach a state (e.g., Invalid) in which it cannot possibly receive any new forwarded requests. If that is not the case, ProtoGen can limit the number of transactions that the cache can observe before its own transaction completes (in effect limiting the size of the transient auxiliary state) and simply stall the controller when this *pending transaction limit (L)* is reached.

Immediate Transition and Responses. The solution above, with deferred sending of responses, preserves the SWMR invariant in physical time, but more aggressive designs are possible. A design in which response sending is immediate, and not deferred, still preserves per-location sequential consistency and is compatible with common consistency models, including SC. As each forwarded request arrives, C_0 observes that a new coherence permission epoch for the block begins in *logical time*. (And all of these epochs are after the cache's permission epoch that resulted from T_{own}.)

This design is otherwise identical to the one above with immediate transitions. The only subtlety is that there are situations in which an immediate response is impossible because it requires sending a message whose contents depend on completing T_{own}. For example, if C_0 is awaiting data for T_{own} and the forwarded request demands the same data from C_0, then C_0 must defer sending the response until it has the data to send. In this case, C_0 uses auxiliary state to remember where to send what messages when sending becomes possible. ProtoGen creates that transient auxiliary state for the cache controller.

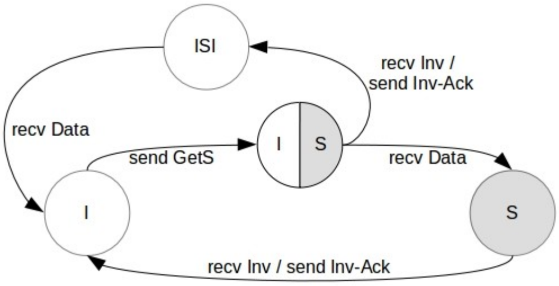

Figure 2. The I to S transition of the MSI protocol. The IS state belongs to the State Sets of $\boldsymbol{I}$ as well as $\boldsymbol{S}$, which is why it is shown in two shades. The ISI state, on the other hand, belongs to only the State Set $\boldsymbol{I}$.

An Example with Immediate Transitions and Responses. Consider cache C_0 and the I to S transition of our MSI protocol. A load to a cache block in state I (a cache miss), leads to C_0 sending a GetS request to the directory and changing the block state to IS. Recall that state IS corresponds to the situation in which a GetS has been sent but the cache is awaiting a response. Hence, it belongs to both $\boldsymbol{I}$ and $\boldsymbol{S}$ State Sets because the cache block can be seen by the directory to be in I or S, depending on whether the GetS has reached the directory.

As shown in Figure 2, if an Invalidation arrives at C_0 in IS, then C_0 knows its own transaction was ordered before the transaction that triggered the Invalidation, because an Invalidation is only possible in stable state S. A stalling protocol would defer processing the Invalidation until C_0 received the response to its own GetS, but we consider the more aggressive protocol with immediate transitions and responses. To avoid stalling, C_0 must process the incoming Invalidation. ProtoGen generates a new transient state which we call ISI. This transient state gets added to State Set $\boldsymbol{I}$, because at this point it is clear to C_0 that its block

[3] If the network has point-to-point ordering then this is trivially guaranteed; forwarded requests will arrive at C_0 in the order the respective transactions were ordered at the directory. If point-to-point ordering is not guaranteed, the directory will have to serialize racing transactions by stalling the second until the first one completes, which again ensures this invariant.

is logically in state I (but still awaiting a data response for its request). C_0's own permission epoch has logically ended (even before the data arrived), and the other cache's permission epoch has logically begun. Also, C_0 immediately sends an acknowledgment to the cache that initiated the transaction that led to the Invalidation. When the data response eventually arrives to complete C_0's transaction, C_0 first serves its stalled load—which is logically part of its *own* epoch—and then transitions to state I.

E. Step 4: Assigning Access Permissions to States

For every state in the protocol, ProtoGen assigns which accesses (load, store, replacement) are allowed in that state. For stable states, this assignment is given in the SSP. For transient states, this assignment is a function of the transient state's initial and final stable states (e.g., a transient state in a transaction from S to M). If the initial and final states both have sufficient permissions for an access, then the access can be performed in the transient state. ProtoGen also has an input parameter that determines whether to permit *any* loads and stores in transient states; recall that permitting this could lead to violations of SWMR in physical time but is still compatible with per-location sequential consistency.

F. Generating Directory Controller

ProtoGen has been described until now from the perspective of generating the cache controller. Although the process of generating the directory controller is quite similar, we now discuss a few issues specific to generating the directory controller.

In general, generating the directory is easier, because the directory has perfect knowledge about the order in which requests are serialized. Even if a directory entry is in a transient state (e.g., in an MSI protocol, a GetS that arrives in state M causes the directory to have to wait for data from the owner), the directory is able to trivially deduce that a subsequent request has to be ordered after the current transaction. Unlike with cache controllers, there is no possibility of an arriving request being ordered before one the directory has already seen.

Directory generation, however, poses one unique challenge. Because our directory controller is non-stalling, there can be as much concurrency at the directory as there are caches in the system. (Concurrency at a cache is constrained by its limit of one outstanding transaction per block.) This concurrency at the directory raises the possibility of observing requests in states that would not be possible in atomic protocols. Specifically, our directories can receive Put requests (PutS, PutM, etc.) in *any* state. For example, consider a block in state S at cache C_0 that issues a PutS to the directory. Before that PutS reaches the directory, cache C_1 issues a GetM that reaches the directory. The GetM changes the directory state to M, and the directory sends an Invalidation to C_0. Later, C_0's now-stale PutS arrives at the directory which is in state M; this situation is not possible in an atomic protocol and thus would not appear in any SSP specification. Furthermore, there are scenarios like this for every combination of Put and every state at the directory.

Because there is no good way to specify these scenarios in an SSP—because they do not occur in an SSP—we leverage knowledge of how directory protocols work. A stale Put request "lost" in its race to the directory and the directory knows that the issuer of the Put had its epoch ended by another transaction that was ordered before its Put. For protocols using a subset of the common MOESIF states, it is correct for the directory to simply acknowledge any stale Put request, so that the issuer of the Put can complete its stale transaction. It might also be possible, in some protocols, to update the directory's auxiliary state (e.g., by removing a cache from the Sharer list), but we do not pursue this option; it is a possible optimization, but not required.

G. Putting It All Together

In this section, we describe how the aforementioned steps are put together to generate cache and directory controllers. ProtoGen first preprocesses the SSP and initializes the State Sets (Step 1). Then, for every stable-to-stable transition in the SSP, ProtoGen generates a transient state for each intermediate step in the transition (Step 2). Then, for each of these transient states, ProtoGen performs the process in Step 3 to accommodate the possible concurrency in that transient state. As part of that process, ProtoGen may generate new transient states. ProtoGen repeats Step 3 on all newly generated transient states until either no new ones remain or we reach the pending transaction limit. After all states are generated, ProtoGen assigns access permissions to every state (Step 4).

VI. Evaluation: Protocols Generated with ProtoGen

To experimentally evaluate ProtoGen, we have used it to generate several different protocols with different features and different system model assumptions. Unlike traditional architectural evaluations that seek to show improvements in performance, power, etc., this evaluation seeks rather to show that ProtoGen can successfully generate protocols that are identical–and, in some cases, arguably superior—to existing protocols.

A. Stalling Protocols

In our first set of experiments, we used ProtoGen to generate several stalling protocols from Sorin et al.'s primer [6]. The primer includes specifications of concurrent MSI, MESI, and MOSI protocols, all of which are stalling. We developed an SSP for each of these protocols—SSPs that are are vastly simpler than the specifications in the primer—and ProtoGen produced concurrent versions of these protocols. For all three of these protocols, ProtoGen generated the same cache controller specifications as in the primer, and the directory controllers were also identical except for one trivial difference for the MSI and MESI directories.[4] All of the protocols passed Murφ verification of SWMR and deadlock freedom with three caches, which is the most caches that Murφ can handle without exhausting memory. The results in this section are perhaps unsurprising but they are reassuring.

B. Non-Stalling Protocols

To test ProtoGen's ability to generate new directory protocols with even more concurrency, we used ProtoGen to generate non-stalling versions of the MSI, MESI, and MOSI protocols from the previous subsection. It is worth noting that the protocols generated were fairly non-trivial with 18-20 states and 46-60 transitions. There are no non-stalling MESI and MOSI protocols in the primer (or specified completely elsewhere), so there are no comparisons to be made. There is, however, a non-stalling MSI protocol in the primer, and we compare our generated protocol to it.

In Table VI, we highlight some differences between our generated protocol and the protocol in the primer. Entries in **bold** are related to the protocol generated by ProtoGen. Where ProtoGen shows a different behavior than the primer's protocol, the transition related to the primer's protocol is crossed out. Two interesting differences are observable. First, the generated protocol is more aggressive, i.e., stalls less often. Even though the primer's protocol is "non-stalling", it still stalls in some complicated situations (e.g., if a $Fwd\text{-}GetS$ or $Fwd\text{-}GetM$ arrives in the states IM^{AD} and SM^{AD}); our generated protocol does not, since it possesses the additional transient states $IM^{AD}S$, $IM^{AD}I$, $IM^{AD}SI$ *and* $SM^{AD}S$. Second, ProtoGen was able to merge some states that were kept separate in the primer like $IM^{\mathrm{A}}S$ = $SM^{\mathrm{A}}S$, $IM^{\mathrm{A}}SI$ = $SM^{\mathrm{A}}SI$, and $IM^{\mathrm{A}}I$ = $SM^{\mathrm{A}}I$.

Verifying non-stalling protocols with a model checker is difficult, because non-stalling protocols tend to enforce SWMR in logical time and not physical time. A model checker seeks to determine whether an invariant is true in the entire reachable state space of the system, and specifying a logical time invariant can be difficult. (State space here refers to all possible states of the entire system, not just the possible coherence states of a given block of memory.) We use Murφ to verify that our protocols do enforce physical time SWMR except in one well-known situation, which is when they perform a single load or store for a transaction whose epoch ended before the data arrived. For example, if a cache issues a GetS to go from I to S so it can perform a load, there is the possibility of an Invalidation arriving while the block is still in state IS; the block transitions to ISI and the cache seemingly fails to perform its load. This is a well-known livelock pitfall, and the common and correct solution is to allow one access (load or store) in physical time after the invalidation [25] (as in the protocol generated by ProtoGen). This access *logically* occurs before the block is invalidated.

C. An MSI Protocol for an Unordered Network

The MSI protocols we have discussed already were designed to work correctly on interconnection networks with point-to-point order. Point-to-point order—which means that, if node A sends two messages to node B, they arrive in the order in which they were sent—makes protocol design easier by eliminating several possible race conditions that could otherwise occur.

To test ProtoGen, we developed the SSP for an MSI protocol that does not rely upon point-to-point order. This protocol adds extra "handshaking" messages to handle the races that arise. Specifying the SSP for this protocol was not much more difficult than for the MSI protocol that relies upon ordering, even with the extra handshaking messages. ProtoGen generated the concurrent protocol from the SSP and thus saved us from having to manually deal with this complexity.

D. TSO-CC

TSO-CC [7] is a recently developed coherence protocol that is tailored for use in systems that support the TSO memory consistency model. Conventional protocols are designed to support *any* consistency model and thus conservatively avoid any behavior that could violate sequential consistency (SC), e.g., by enforcing SWMR in physical time. TSO-CC, by contrast, exploits the relaxed nature of TSO to avoid sharer tracking. In doing so, it breaks physical time SWMR but honors TSO.

The TSO-CC paper is accompanied by a complete protocol specification with concurrency that is designed to work correctly even if the network is unordered. We wanted to see if we could use ProtoGen to generate a complete TSO-CC protocol with concurrency but leveraging point-to-point ordering.

The first step was to develop an SSP specification that can leverage point-to-point ordering. This was reasonably straightforward given the complete TSO-CC specification; it was a question of selecting the stable state transitions and eliminating "handshakes" to exploit point-to-point ordering.

We then used ProtoGen to generate the complete protocol with concurrency. Using the verification methodology of Banks et al. [26], we verified that our complete protocol correctly enforces TSO. The key takeaways from this study are twofold. First, ProtoGen can be used to generate unconventional protocols such as TSO-CC. Second, it also showcases ProtoGen's utility in transforming a complex protocol and making it work for a different system model. Our protocol modification was easy to make at the SSP level, whereas it would have been much more difficult to generate

[4] ProtoGen split a state to more precisely track the sharer list in one rare situation.

Table VI
MSI NON-STALLING PRIMER VS. PROTOGEN

	Load	Store	Replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data (ack=0)	Data (ack>0)	Inv-Ack	Last Inv-Ack
I	send GetS to Dir/IS^{D}	send GetM to Dir/IM^{AD}									
IS^{D}	stall	stall	stall			send Inv-Ack to Req/IS^{D}I		-/S			
IS^{D}I	*stall*	*stall*	*stall*					*-/I*			
IM^{AD}	stall	stall	stall	~~stall~~ **-/IM^{AD}S**	~~stall~~ **-/IM^{AD}I**			-/M	-/IM^{A}	ack–	
IM^{A}	stall	stall	stall	-/IM^{A}S	-/IM^{A}I					ack–	-/M
IM^{A}S = **SM^{A}S**	stall	stall	stall			send Inv-Ack to Req/IM^{A}SI				ack–	send Data to Req and Dir/S
IM^{A}SI = **SM^{A}SI**	stall	stall	stall							ack–	send Data to Req and Dir/I
IM^{A}I = **SM^{A}I**	stall	stall	stall							ack–	send Data to Req/I
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^{A}			send Inv-Ack to Req/I					
SM^{AD}	hit	stall	stall	~~stall~~ **-/SM^{AD}S**	~~stall~~ **-/IM^{AD}I**	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^{A}	ack–	
SM^{A}	hit	stall	stall	~~-/SM^{A}S~~ **-/IM^{A}S**	~~-/SM^{A}I~~ **-/IM^{A}I**					ack–	-/M
M	hit	hit	send PutM + Data to Dir/MI^{A}	send Data to Req and Dir/S	send Data to Req/I						
IM^{AD}S	*stall*	*stall*	*stall*			*send Inv-Ack to Req/IM^{AD}SI*		*send Data to Req and Dir/S*	*-/IM^{A}S*	*ack–*	
IM^{AD}I	*stall*	*stall*	*stall*					*send Data to Req/I*	*-/IM^{A}I*	*ack–*	
IM^{AD}SI	*stall*	*stall*	*stall*					*send Data to Req and Dir/I*	*-/IM^{A}SI*	*ack–*	
SM^{AD}S	*hit*	*stall*	*stall*			*send Inv-Ack to Req/IM^{AD}SI*		*send Data to Req and Dir/S*	*-/IM^{A}S*	*ack–*	
MI^{A}	stall	stall	stall	send Data to Req and Dir/SI^{A}	send Data to Req/II^{A}		-/I				
SI^{A}	stall	stall	stall			send Inv-Ack to Req/II^{A}	-/I				
II^{A}	stall	stall	stall				-/I				

a complete TSO-CC protocol at the concurrent protocol level that is able to leverage point-to-point ordering.

E. Discussion

Our current ProtoGen implementation has not been optimized for performance, but was designed for flexibility during the development process. Nevertheless, runtimes are always well less than one second on an Intel i5.

Although one could argue that many of the protocols in this section already existed and generating them is not practically useful, automatic generation is still far faster and less error-prone than designing protocols by hand. Moreover, it is exciting to observe that ProtoGen could handle an unconventional protocol like TSO and uncover additional concurrency in non-stalling protocols.

VII. CONCLUSIONS

We have developed ProtoGen to help architects design directory cache coherence protocols. ProtoGen simplifies the design process by requiring the architect to specify only the stable state protocol with atomic transactions. At its heart, ProtoGen is a method for refining an atomic specification into a non-atomic implementation. In contrast with other general techniques, ProtoGen exploits domain knowledge about how directory protocols work—most importantly, the fact that coherence transactions serialize at the directory—to enable it to generate highly-concurrent non-blocking protocols

We have shown, for a variety of protocols, that ProtoGen successfully generates the finite state machines for cache and directory controllers. Furthermore, ProtoGen has generated protocols with at least as much concurrency as those found in existing protocols, suggesting that generated protocols need not sacrifice performance compared to manually designed protocols.

Acknowledgments

We would like to thank our shepherd, Michael Pellauer, and the anonymous reviewers for their valuable comments. This work was supported by EPSRC under grants EP/M027317/1 and EP/L01503X/1 to The University of Edinburgh and by the National Science Foundation under grant CCF-142-1167. Daniel Sorin was supported by fellowships from the Royal Academy of Engineering, the Scottish Informatics & Computer Science Alliance, and the Leverhulme Trust.

References

[1] "Coherency was broken and manually disabled in galaxy s4," https://www.anandtech.com/show/7164/samsung-exynos-5-octa-5420-switches-back-to-arm-gpu, note = Accessed: 2018-04-10.

[2] N. Dave, M. C. Ng, and Arvind, "Automatic synthesis of cache-coherence protocol processors using bluespec," in *MEMOCODE*, 2005.

[3] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," in *PLDI*, 2013.

[4] C. J. Banks, M. Elver, R. Hoffmann, S. Sarkar, P. Jackson, and V. Nagarajan, "VerC3: A library for explicit state synthesis of concurrent systems," in *DATE*, 2018.

[5] D. L. Dill, "The Murphi Verification System," in *CAV*, vol. 1102, 1996.

[6] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[7] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, 2014.

[8] "Bluespec system verilog," http://bluespec.com/, note = Accessed: 2018-03-30.

[9] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[10] J. Staunstrup and M. R. Greenstreet, "From high-level descriptions to VLSI circuits," *BIT*, vol. 28, no. 3, 1988.

[11] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," *IEEE Micro*, vol. 19, no. 3, 1999.

[12] N. Dave, Arvind, and M. Pellauer, "Scheduling as rule composition," in *MEMOCODE*, 2007.

[13] M. Karczmarek and Arvind, "Synthesis from multi-cycle atomic actions as a solution to the timing closure problem," in *ICCAD*, 2008.

[14] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *HPCA*, 2011.

[15] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Automatic completion of distributed protocols with symmetry," in *CAV*, 2015.

[16] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," *PACT*, 2011.

[17] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *PACT*, 2012.

[18] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO*, 2010.

[19] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, Sep. 1979.

[20] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: An approach for extending smps," in *ASPLOS*, 2000.

[21] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory coherence in the age of multicores," in *ICCD*, 2011.

[22] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *HPCA*, 2013.

[23] S. Chandra, B. Richards, and J. R. Larus, "Teapot: A domain-specific language for writing cache coherence protocols," *IEEE Trans. Software Eng.*, vol. 25, no. 3, 1999.

[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, 2005.

[25] J. Kubiatowicz, D. Chaiken, and A. Agarwal, "Closing the window of vulnerability in multiphase memory transactions," in *ASPLOS*, 1992.

[26] C. J. Banks, M. Elver, R. Hoffmann, S. Sarkar, P. Jackson, and V. Nagarajan, "Verification of a lazy cache coherence protocol against a weak memory model," in *FMCAD*, 2017.