




Semi-open-state testing for in-silicon coherent interconnects

Jasmin Schult^{}, Ben Fiedler^{}, David Cock^{}, Timothy Roscoe^{}

ETH Zürich, Zürich, Switzerland

firstname.lastname@inf.ethz.ch

Abstract—In this paper, we extend open-state conformance testing from Mealy FSM specifications to implementations where only a subset of states are observable. We show that the classical transition tour can be used to completely test such implementations for conformance, including unobservable states, under the assumption that in the specification, any trace from an unobservable state eventually reaches an observable state – i.e. that the observable states form a feedback vertex set. Complete transition tour testing can efficiently test for many conformance relations when they are appropriately formulated. Generalized-quasi-reduction (GQR) is useful for protocol testing because it allows for partial, non-deterministic specifications, but establishing GQR in the general setting is complex and expensive. We show a relation that implies GQR and is practical for transition tour testing.

Our setting of partial state observability applies to an important class of protocol implementations in modern hardware: cache coherence. These protocols are nearly universal in multi-processor systems, and are notoriously difficult to verify both at the specification and implementation level. We show that their structure lends them naturally to complete open-state testing under this extended definition. During design, coherence protocols are elaborated from an FSM of *stable states* with atomic, observable transitions by the addition of a large number of unobservable *transient states* to handle concurrency, including out-of-order and interleaved message delivery. We demonstrate that a real in-silicon implementation on the Cavium ThunderX-1 CN88XX CPU has exactly the required characteristics and we establish the GQR conformance relation against a specification of its inter-socket coherence protocol.

I. INTRODUCTION

In this paper, we introduce *semi-open-state testing*, a test setting where an implementation with partially visible states is tested from a finite state machine model of its behavior to establish a conformance property like GQR. We show that under a set of additional requirements that we impose on semi-open-state testing, complete visibility of the implementation’s states is achieved, thus allowing simpler *open-state testing* techniques [1] to be employed instead of the more expensive, general complete Finite State Machine (FSM) testing methods [2]. This work arises out of our analysis of *cache coherence protocols* in the context of the Enzian [3] project. We show that this practically-important class of protocols match the requirements of semi-open-state testing by demonstrating a complete, online validation that the native protocol of the server-class CN88XX (ThunderX-1) CPU is GQR-conformant to its specification. This testing is conducted live against the actual silicon implementation.

Semi-open-state testing is motivated by the structure of hardware cache coherence protocols, particularly in emerging standards such as CXL [4]. These protocols consist of a set of *stable* states that are generally visible via debug mechanisms, plus a larger number of intermediate *transient* states added to handle the effects of concurrency between nodes e.g. message reordering. A typical protocol has 3–5 stable states, and 10–100 transient states. The degree of reordering is bounded in practice, and thus a transient state will lead back to a stable state after a finite number of messages. Our insight is to label these intermediate states with the last stable state and this finite, observable IO trace, thereby reducing the task to standard open-state testing. Open-state testing methods are both simpler and less expensive, which greatly increases their practicality for testing real hardware.

Formally, this work concerns the *complete conformance testing* [5] of implementations against *Mealy FSM* specifications. Existing work leverages either general complete FSM testing or *open-state testing*. The general complete testing methods only assume an upper bound m on the number of states in the implementation, hence they are referred to as *m-complete* methods. They are comparatively heavy-weight and worst-case exponential in m . Open-state testing, on the other hand, derives large gains in efficiency and implementation complexity from its stronger assumptions on the system under test. Chief among these is that all implementation states are *visible*. The first challenge in applying open-state techniques to coherence protocols is thus the invisibility of transient states. We overcome this by developing semi-open-state testing, the first key contribution of this paper.

The second challenge is in identifying a conformance relation that matches the characteristics of a coherence protocol. We settle on GQR principally for its ability to permit partial and non-deterministic specifications, which is essential in a practical protocol specification to leave sufficient freedom for implementors to optimize their designs. GQR is presented in an *m-complete* setting, and we know of no existing open-state-compatible formulation in the literature. Our second contribution is thus the open-state-testable relation GQR_{open} , which we prove is testable under the assumptions of semi-open-state testing and sufficient to establish standard GQR for the systems we consider.

In section II, we introduce both conformance testing from Mealy FSMs and GQR as used in existing literature. We develop semi-open-state testing in section III, enumerating

the assumptions needed to extend open-state testing to semi-visible implementations. In section IV we explain the function and characteristics of cache coherence protocols, why semi-open-state testing is applicable to them and why the emergence of coherent interconnect standards necessitates such testing. We demonstrate how this approach performs in practice in section V, by applying semi-open-state testing to the cache coherence implementation of the ThunderX-1, which is deployed as part of the Enzian platform. Finally, we conclude and outline future work in section VI.

II. BACKGROUND AND RELATED WORK

A. Notation & Definitions

We consider Mealy-type IO state machines whose output depends both on state and current input, and adapt the notation of Hierons [6]. A machine is a tuple $M = (S, s_0, X, Y, h)$ where:

- S is the finite set of states.
- $s_0 \in S$ is the initial state.
- X is the finite input alphabet.
- Y is the finite output alphabet.
- $h \subseteq (S \times X) \times (S \times Y)$ is the transition relation.

Where necessary, a subscript identifies the associated machine, e.g. h_{SPEC} for the transition relation of machine SPEC.

If $((s, x), (s', y)) \in h$ then on receiving input x when in state s , the machine may transition to state s' while producing output y . $h(s, x)$ denotes the image of $\{(s, x)\}$ under h , i.e. the allowed transitions for a given state and input.

We assume that *all states in S are reachable* from the machine's initial state, since unreachable states are not relevant to the observable behavior of the machine.

We assume that M is *observably non-deterministic*: The target state is completely determined by the observed input and output. Equivalently, the outputs of any two distinct reactions in $h(s, x)$ must differ:

$$\{(s', y'), (s'', y'')\} \subseteq h(s, x) \implies s' = s'' \vee y' \neq y'' \quad (1)$$

We denote the set of inputs x for which some transition of machine M is defined from state s by

$$\Omega_M(s) = \{x | ((s, x), (_, _)) \in h_M\}$$

M is *completely specified* if $\forall s. \Omega_M(s) = X$ i.e. its behavior is specified for all inputs in all states.

Y contains the distinct special symbols \sim and \perp representing no output and failure. \sim is allowed for both specification and implementation, but \perp only for an implementation. \sim in a specification requires that an implementation produce no output, and \sim in an implementation is assumed to be observable (e.g. by timeout [7]). \perp does not satisfy any specification. With these additions we may assume that any implementation is completely specified.

Every machine step consumes exactly one input, and produces exactly one output. A length- n trace of the machine records both state transitions and the corresponding input/output pairs:

$$((s, x_1), (s_1, y_1)), \dots, ((s_{n-1}, x_n), (s_n, y_n))$$

The corresponding *IO trace* is the projection containing only the message pairs:

$$x_1/y_1, \dots, x_n/y_n$$

We write $t.x/y$ for the IO trace t extended with the IO transition x/y and t_1t_2 for trace concatenation.

The language $L_M(s)$ is the set of permitted IO traces beginning with machine M in state s , and for any IO trace $t \in L_M(s)$ of an observably non-deterministic machine,

$$\text{AFTER}_M(s, t)$$

is the unique final state s' that machine M reaches after observing trace t beginning in state s . We omit the state s if it corresponds to the initial state: $L_M := L_M(s_0)$ and $\text{AFTER}_M(t) := \text{AFTER}_M(s_0, t)$.

Note that

$$\text{AFTER}_M(s, (t')) = \text{AFTER}_M((\text{AFTER}_M(s, t)), t') \quad (2)$$

B. Conformance Testing against Mealy Machines

Testing an implementation against a specification is a well-studied problem, known as conformance testing, model-based testing, and fault detection in the literature. Different techniques exist to test against a variety of formal models; This paper considers conformance testing against Mealy FSMs.

Establishing a relation between a real-world implementation and an abstract specification SPEC begins with the *testability hypothesis* [8]. For Mealy FSM specifications, this includes that the behavior of the implementation can be captured by an unknown, abstract Mealy FSM denoted by IMPL.

For IMPL to be testable even if its behavior is non-deterministic requires us to assume (at least) *weak fairness*: If the implementation is presented with input x in state s enough times, it will eventually take every possible transition (to some state s' with output y).

Existing work [5]–[7], generally considers IMPL to be a *completely specified, observably non-deterministic* Mealy FSM. Complete specification captures the practical reality that a real-world implementation cannot refuse an input provided by its environment and so always defines some reaction, even if this is just failing or producing no output. These are the \sim and \perp output symbols already introduced. Further, any completely-specified Mealy FSM can be transformed into an observably non-deterministic equivalent via the standard NFA–DFA construction [9].

Requirement 1. (*Testability Hypothesis*) *In practice, the testability hypothesis requires a suitable I/O adaptor to translate between the implementation's concrete and the specification's abstract I/O. Aarts et al. formalize the requirements of such an adaptor in the context of state machine learning [10]. Furthermore, the abstract implementation needs to be input-driven by the input alphabet of SPEC.*

For the fairness assumption to hold in practice, it is sufficient that the implementation's non-determinism originate only from true randomization. In particular, the implementation's non-determinism must not originate from concurrent

processing of inputs resulting in different outcomes (which would require testing to generate unknown relative input timings to explore all such outputs). Equivalently, the implementation's processing of inputs must be linearizable.

It is important to ascertain that a real-world implementation exhibits these properties before testing it against a Mealy specification. We revisit these assumptions in the context of coherence protocols in section IV.

Testing seeks to establish a *conformance relation* between SPEC and IMPL. In this case we work with GQR as defined by Hierons [6]. GQR allows partial specifications, i.e. it does not require a transition in response to every input in every state. This is necessary for protocols, which are typically sparse; only a small set of inputs are expected in a particular state. GQR interprets non-determinism in the specification as implementation choice. An implementation is a *generalized quasi reduction* of its specification if all its outputs are valid choices offered by the specification, so long as only inputs defined by the specification are provided to it. More formally,

Definition 1 (Generalized-Quasi-Reduction (GQR)).

$$\forall t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}, x \in \Omega_{\text{SPEC}}(\text{AFTER}_{\text{SPEC}}(t)). \\ \{y|t.x/y \in L_{\text{IMPL}}\} \subseteq \{y|t.x/y \in L_{\text{SPEC}}\} \quad (3)$$

This definition simplifies the original in two ways: Firstly, as our SPEC is observably nondeterministic, $\text{AFTER}_{\text{SPEC}}(t)$ is unique, and thus we need only quantify over its inputs and not over the set of possible states as well. Secondly, as IMPL is completely specified we can drop the first condition, which requires a corresponding implementation transition.

Conformance testing methods are either *complete* or *incomplete*. Complete methods can formally establish a conformance relation, whereas incomplete methods increase the confidence that a relation holds, but without formal guarantees. We consider only complete conformance testing.

The difficulty of complete conformance testing depends on the additional assumptions made on the implementation. Without any assumptions, the IO behavior of even a finite machine may be infinite, and thus impossible to completely test with finite methods.

Complete testing generally assumes a known upper bound m on the number of implementation states, giving *m-complete testing* [6]. The number of transitions that must be tested is exponential in m , as shown by Moore [11]. In particular, it is insufficient to cover all transitions of the specification once [2]. Note that this is true even if the implementation is assumed to have at most as many states as the implementation, since the implementation may enter the wrong state after a transition. This is due to the implementation state not being visible: it can only be inferred from the future I/O traces of the implementation.

If the state is visible, the complexity decreases dramatically. This is *open-state testing* [1] or testing with a reliable *status message* [5], [12]. The finite representations of specification and implementation can be compared directly, instead of

reasoning over all their (potentially) infinitely many I/O traces. Complete testing against a specification reduces to an online traversal of its transition graph, with sufficient repetitions to account for non-determinism. Bourdonov et al. [1] describe a traversal algorithm that can cope with detours caused by unfavorable choices of non-determinism in the implementation.

Open-state testing is not only simpler to implement but also more efficient than m -complete testing. Sidhu et al. [13] show that transition traversal generates shorter test cases on average compared to m -complete methods for deterministic, complete specifications; this also holds even if the bound m is equal to the number of states in the specification. Open-state testing is much more efficient than the general methods used to establish GQR, as testing against partial, non-deterministic specifications is considerably more difficult [6], [14]. Furthermore, open-state testing is able to detect if an implementation possesses any number of additional states with respect to the specification without an increase in testing complexity, whereas we pay an exponential tax to do the same for m -complete methods.

In the following sections, we show that m -complete testing for GQR can be reduced to open-state testing, even if only a subset of states are actually visible (section III), and demonstrate that the necessary assumptions apply to an important class of practical protocols: those for hardware cache coherence (section IV).

III. SEMI-OPEN-STATE TESTING AND GQR_{OPEN}

Requirement 2. (*Visible States*) We require that a finite subset of reachable implementation states, including the initial state, are known a priori:

$$s0_{\text{IMPL}} \in S_{\text{known}} \subseteq S_{\text{IMPL}}$$

These states are fully observable—we see whether or not the implementation is in such a state and if so, which one. All others are only indirectly observable by their IO behavior.

We thus obtain a partial labelling function, λ , from traces to states:

$$\text{AFTER}_{\text{IMPL}}(t) = s \wedge s \in S_{\text{known}} \implies \lambda(t) = s \quad (4)$$

We label the unknown states by reference to the last known state. For any trace t from s_0 , take the longest t_K such that

$$t_K t_U = t \quad \lambda(t_K) \in S_{\text{known}}$$

We thus obtain a complete labelling of traces, $\hat{\lambda}$, with the tuple of last known state and IO trace since that state:

$$\hat{\lambda}(t) = (\lambda(t_K), t_U) \quad (5)$$

As IMPL is observably nondeterministic, the trace uniquely defines the final state, which is thus observable given the observed label of the IO trace to the last known state (t_K), plus the IO trace through all subsequent unknown states (t_U).

For any known state s_K , a corresponding trace t_K , and a continuation t_U such that $t := t_K t_U \in L_M$ (or equivalently

$\hat{\lambda}(t) = (s_K, t_U)$, the state s corresponding to trace t 's label matches the machine's actual state after t :

$$\begin{aligned}
s &:= \text{AFTER}_{\text{IMPL}}(\hat{\lambda}(t)) \\
&= \text{AFTER}_{\text{IMPL}}(\lambda(t_K), t_U) && \text{by (5)} \\
&= \text{AFTER}_{\text{IMPL}}(\text{AFTER}_{\text{IMPL}}(t_K), t_U) && \text{by (4)} \\
&= \text{AFTER}_{\text{IMPL}}(t_K t_U) && \text{by (2)} \\
&= \text{AFTER}_{\text{IMPL}}(t) && \text{(6)}
\end{aligned}$$

Hence $\hat{\lambda}$ is equivalent to labelling states with the full trace from s_0 which, by observable nondeterminism, uniquely identifies the state. $\hat{\lambda}$ thus allows a tester to identify the implementation state after any trace, providing the complete visibility of the implementation's states needed for open-state testing.

To complete the reduction to open-state testing, the states of SPEC need to match the observations returned by $\hat{\lambda}$. To preserve the finiteness of S_{SPEC} , we need to impose the following requirement on the specification that we wish to test against:

Requirement 3. (*Feedback Vertex Set*) We require that the known states form a feedback vertex set¹ in the specification's transition graph, and every trace through only unknown states is finite. We further require that there exists some global bound, c , on the length of all such traces.

The transformation of the original specification to a *semi-open testable* SPEC entails converting the DAGs between the stable states in its transition graph to trees by duplicating states as necessary. This transformation may therefore increase the number of transitions and states to $\mathcal{O}(|S_{\text{known}}| \cdot (|X| \cdot |Y|)^c)$ in the worst case, depending on the shape of the transition graph. We therefore need to carefully examine the specification to determine if open-state testing remains practical and preferable to m -complete testing:

Requirement 4. (*Practicability*) The bound c and the shape of the original specification should be carefully examined to determine if the size of the corresponding semi-open testable SPEC remains manageable.

With the resulting transformed SPEC, our reduction of semi-open-state testing to open-state testing is complete.

We now shift our attention to how open-state testing can be leveraged to establish GQR. A relation is *open-state testable* if, assuming that the tester can drive the machine's input and observe both its state and output, it is possible to determine if a specification and implementation lie in the relation by driving the implementation through all input transitions in the specification a finite number of times (to account for non-determinism). This implies that an *open-state testable* relation can be established by a graph traversal of the specification, as in the algorithm of Bourdonov [1].

GQR as formulated in Definition 1 is not directly *open-state testable*. It permits a single specification state to be

implemented as multiple states that are distinguishable only by their inconsistent choices of non-deterministic options. For a detailed discussion of this phenomenon, we refer to the study of the classical reduction relation by Petrenko et al. [14]. In the open-state setting, the observations of these multiple implementation states will fail to match the single state in the specification, causing the test to fail even if the implementation is GQR in the general sense.

We therefore propose the following *open-state-testable* specialization of Definition 1, which assumes the observability of implementation states:

Definition 2 (Open-State GQR (GQR_{open})).

$$\begin{aligned}
s0_{\text{IMPL}} &= s0_{\text{SPEC}} \wedge \\
&\quad \forall s \in S_{\text{SPEC}}. \forall x \in \Omega_{\text{SPEC}}(s). \\
&\quad (\exists t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}. s = \text{AFTER}_{\text{IMPL}}(t)) \\
&\quad \implies h_{\text{IMPL}}(s, x) \subseteq h_{\text{SPEC}}(s, x) \quad (7)
\end{aligned}$$

Notice that GQR_{open} is comparing states of the implementation and states of the specification for equality (in particular, recall that the transition functions h_{SPEC} and h_{IMPL} map to sets of (next state, output)). This is possible because the states that the implementation adopts are completely visible in the open state setting and can hence act like an additional output of the implementation. Recall that for the semi-open-state testing setting that we have discussed earlier, this visibility is provided by the $\hat{\lambda}$ function.

GQR_{open} implies, inductively, that the states of the implementation that are reached by traces defined by the specification must agree with those of the specification:

Lemma 1.

$$\begin{aligned}
\text{GQR}_{\text{open}} &\implies \forall t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}. \\
&\quad \text{AFTER}_{\text{IMPL}}(t) = \text{AFTER}_{\text{SPEC}}(t)
\end{aligned}$$

Initially,

$$\text{AFTER}_{\text{IMPL}}(\square) = s0_{\text{IMPL}} = s0_{\text{SPEC}} = \text{AFTER}_{\text{SPEC}}(\square)$$

Take any

$$t.x/y \in L_{\text{SPEC}} \cap L_{\text{IMPL}} \quad (8)$$

such that

$$s := \text{AFTER}_{\text{IMPL}}(t) = \text{AFTER}_{\text{SPEC}}(t) \quad (9)$$

Since $t.x/y \in L_{\text{SPEC}} \cap L_{\text{IMPL}}$, by the definition of h_M :

$$x \in \Omega_{\text{SPEC}}(s) \quad (10)$$

$$(\text{AFTER}_{\text{IMPL}}(t.x/y), y) \in h_{\text{IMPL}}(s, x) \quad (11)$$

$$(\text{AFTER}_{\text{SPEC}}(t.x/y), y) \in h_{\text{SPEC}}(s, x) \quad (12)$$

Given (8,9, and 10), GQR_{open} (7) yields:

$$h_{\text{IMPL}}(s, x) \subseteq h_{\text{SPEC}}(s, x)$$

or, the implementation's transitions are a subset of the specification's.

¹A feedback vertex set of a directed graph G is a set of vertices whose removal transforms G into a directed acyclic graph.

Combined with (11) we have that

$$(\text{AFTER}_{\text{IMPL}}(t.x/y), y) \in h_{\text{SPEC}}(s, x)$$

or, the final implementation state is among those of the specification with the same observable IO behaviour.

The actual specification state must also be in this set (12), and thus by the definition of observable non-determinism (1) must be equal to the implementation state:

$$\text{AFTER}_{\text{IMPL}}(t.x/y) = \text{AFTER}_{\text{SPEC}}(t.x/y)$$

□

Lemma 2. GQR_{open} is open-state testable.

Definition 2 applies to the set of implementation states reachable by a trace also accepted by the specification. This set is finite. Assuming weak fairness (Requirement 1), we will eventually observe every possible y for each s and x . Thus by repeatedly traversing every input transition we will terminate, having exhaustively tested the subset relation. □

Lemma 3. $\text{GQR}_{\text{open}} \implies \text{GQR}$

Using Definition 1 (GQR) take $t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}$, $x \in \Omega_{\text{SPEC}}(\text{AFTER}_{\text{SPEC}}(t))$ and y such that $t.x/y \in L_{\text{IMPL}}$.

Since $t, t.x/y \in L_{\text{IMPL}}$,

$$\exists s'. (s', y) \in h_{\text{IMPL}}(\text{AFTER}_{\text{IMPL}}(t), x) \quad (13)$$

Moreover, from GQR_{open} we have, by Lemma 1

$$\text{AFTER}_{\text{SPEC}}(t) = \text{AFTER}_{\text{IMPL}}(t)$$

Thus, by GQR_{open} (7)

$$h_{\text{IMPL}}(\text{AFTER}_{\text{IMPL}}(t), x) \subseteq h_{\text{SPEC}}(\text{AFTER}_{\text{SPEC}}(t), x)$$

Together with (13) we have

$$(s', y) \in h_{\text{SPEC}}(\text{AFTER}_{\text{SPEC}}(t), x)$$

Thus, since $t \in L_{\text{SPEC}}$:

$$t.x/y \in L_{\text{SPEC}}$$

□

In this section we have presented semi-open-state testing for semi-visible machines, and GQR_{open} : a sufficient, semi-open-state-testable condition for GQR to hold on the traces of such a machine. By exploiting the observability of the subset of known states and the observable nondeterminism of the implementation, we construct a complete labelling, $\hat{\lambda}$, of implementation states. Given an appropriately shaped specification, this allows us to reduce semi-open-state testing to open-state testing, avoiding the cost of general m -complete testing.

IV. CACHE-COHERENCE PROTOCOL INTEROPERABILITY

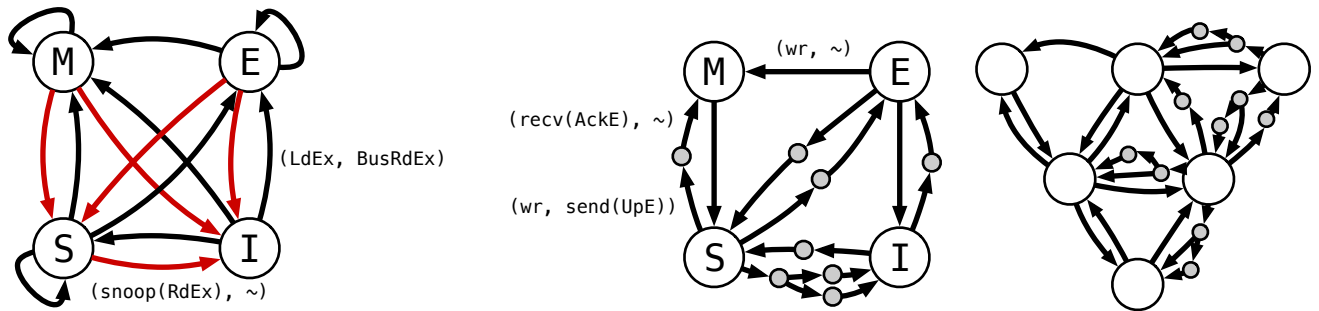
We now turn to a key real-world problem which is highly amenable to semi-open-state testing. Indeed, our motivation to develop the formalism stemmed from a practical problem we faced: how to gain confidence that two different endpoint implementations of an informally-defined and under-specified cache coherence protocol will successfully interoperate. The endpoints of mainstream inter-processor cache coherence protocols turn out to be an excellent match to the requirements for semi-open-state testing.

Modern computers with multiple processor cores rely on caches for performance: a hierarchy of caches holds copies of data from memory (*lines*), and these caches are kept *coherent* by a hardware *cache coherence protocol* which ensures that, at any point in time, all copies of a line that reside in the system's caches are identical [15]. This is a global invariant that must be upheld at all times; the protocol maintains this invariant while serving memory requests (reads and writes) made by different cores. To correctly and efficiently negotiate the simultaneous handling of multiple such requests, the endpoints of modern coherence protocols tend to be large and complex state machines. At the same time, high assurance in the correct operation of these endpoints is required: bugs prevent correct execution of the entire machine, and the performance-critical implementation in silicon means that these bugs can rarely be fixed post-silicon. For this reason, formal methods have long been employed in coherence protocol designs [16], e.g. for verifying the protocol definitions [17], or generating correct-by-construction protocol state machines [18].

Work to date that tests if hardware implementations correctly implement these verified protocol designs has operated on the entire coherent system, rather than on individual protocol endpoints as we are proposing: Kahlouche et al. [19] and Kriouile et al. [20] also generate tests from formal models, but the scale of the system-wide protocol makes complete testing intractable and simulation environments are needed to exert control over the concurrent execution at the protocol endpoints. Consequently, these works have neither addressed the visibility of endpoint states nor leveraged those states to achieve complete testing coverage. Orthogonally, other efforts [21], [22] have integrated additional testing logic into the system implementation to generate test stimuli and to directly check the high-level protocol invariants.

These existing testing methods have worked so far because coherence protocols have generally been specific to a particular processor model, allowing a single hardware team to conduct the design, verification, and implementation of the entire coherent system.

This situation is changing: open, cache-coherent interconnect standards like CXL.cache [4], NVlink [23], CCIX [24], and TileLink [25] attach a range of 3rd-party cache-coherent devices to a computer system. The resulting new and potentially *different* protocol endpoint implementations of these devices must be able to interoperate with each other and achieve global coherence in flexible system compositions.



(a) A snoop-based, symmetric MESI protocol (b) Asymmetric specs for 2-node directory-based MESI; the remote side (left) tracks only M - E - S - I states, and the home side (right) tracks local and remote states.

Fig. 1: Mealy machines for snoopy and directory-based MESI.

This development motivates both the formal specification of protocol endpoints in such standard, and the means to efficiently and exhaustively test their resulting in-silicon implementations. More concretely, formal specification allows the desired system-level properties to be verified on the abstract compositions of the standard’s endpoints. A complete test method then allows those system-level properties to be transferred from the abstract to the composition of successfully tested endpoint implementations. To apply to a standard’s complex multi-vendor ecosystem, this test method must operate on the in-silicon implementation only, without requiring access to additional information such as internal documentation or design sources.

Given that the requirements for semi-open-state testing are met, our proposed testing approach is applicable to this setting: a successful test verdict guarantees complete conformance of an in-silicon coherence endpoint to its formal Mealy FSM specification. Semi-open-state testing does not require any additional information beyond access to the in-silicon implementation, and can be naturally integrated into the usual compliance testing workshops conducted for hardware interconnect standards like PCI Express (PCIe).

We will now discuss why the requirements of semi-open-state testing can be met by protocol endpoints of cache coherent interconnect standards. To this end, we take a closer look at the nature of these endpoints. We then argue why the requirements of semi-open-state testing constitute reasonable restrictions on these endpoints and can therefore be imposed on vendor implementations by the standard.

A. Directory-based cache coherence protocols

Basic textbook coherence protocols tend to be *snoopy*: each node can observe the operations performed by all other nodes instantaneously. The classic example is MESI, which associates one of four different states with each line: M (*odified*), E (*xclusive*), S (*hared*) or I (*nvalid*). The M and E states imply that the cache holds the only valid copy (dirty or clean resp.) of the data and reads and writes can be performed locally on it without coordination. State S implies the copy is valid and clean but may exist in other caches, requiring coordination for writes. A snoopy MESI protocol endpoint can thus be specified

with the Mealy FSM in Figure 1a. Transitions are defined on inputs corresponding to local memory requests or snooped-on remote bus transactions, while outputs are initiated bus transactions.

The shared bus required for snoopy protocols does not scale well, and so in practice most real inter-die implementations are *directory-based*, including those used by coherent interconnect standards. These protocols track the cache line status of all participating nodes in a *directory* held at the line’s *home node* (typically where the main memory for the line is attached), and coordinate with explicit point-to-point messages instead of bus snooping.

This makes the protocol endpoints asymmetric: remote nodes have the same *stable* states (e.g. M , E , S , I) as the snoopy protocol, but the home directory needs stable states that reflect the system-wide state combinations. Moreover, the use of point-to-point messages requires additional *transient states* (sometimes hundreds) in order to cope with all possible interleavings of messages and actions on each node, including conflicting concurrent transactions and message reordering. The resulting two Mealy FSMs are therefore more complex (Figure 1b).

B. Connection to semi-open-state testing

The endpoints of directory-based coherence protocols are amenable to semi-open-state testing: the special visible states in the implementation correspond to the stable protocol states, and the main assumptions we require do hold.

Requirement (1) Testability Hypothesis: A valid *I/O adaptor* can abstract the data in a cache line, assembly load and store instructions, and the format of coherence messages, retaining only the message and software request types. The abstracted behavior of the in-silicon implementation can be *driven* with respect to the inputs of the spec, which requires that the protocol-relevant state does not change except in response to such inputs. Notice that our state machines only reason about a single cache line; silicon implementations generally do indeed treat each line independently [16], although some dependencies may be introduced if the implementation relies on particular message reorderings [26]. For the standard fairness assumption, we further require the implementation’s

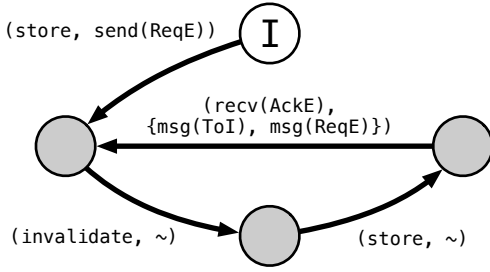


Fig. 2: A cycle of transient states in a remote node, produced by its software repeating concurrent invalidates and reads.

processing of inputs to be *linearizable*. This is usually the intended behavior of a protocol endpoint; whether linearizable processing is achieved needs to be validated separately.

Requirement (2) Visible stable cache line states: Although not used by most programmers, it turns out that existing hardware generally provides precisely this property to software via facilities intended for performance analysis and low-level debugging. An example is the processor we test in section V. Furthermore, the initial state of the protocol naturally corresponds to the *stable Invalid state*, and is therefore among the visible states, as required.

Requirement (3) Stable cache states form a feedback vertex set: To achieve this, our specification must *exclude input buffering* and the remaining *pure protocol processing* must exhibit the feedback vertex property. Without the exclusion of input buffering, a continuous stream of stores and invalidations on the remote node could yield a cycle of transient states, as shown in Figure 2: as soon as coordination with the home node completes, a request is immediately replaced by its buffered successor, and thus no intermediate stable state is visible.

Real implementations do require the buffering behavior we exclude from the specification. However, in practice the buffering only depends on the aspects of the protocol state that the implementation makes directly visible: only entering a stable state serves as a signal to fetch the next request from the buffer. This choice is made by hardware designers precisely to limit complexity of both implementation and validation.

Consequently, we only need pure protocol processing to respect the feedback vertex set property. This requirement might preclude aggressive cache optimizations such as eager replies to requests, or remote-allocate of the line into a remote node’s cache (which may require the remote node to handle an unbounded number of home-enforced caching state upgrades and downgrades while a request of its own is pending). In practice, such features are rare.

Requirement (4) Practicability: Recall that the specification needs to be transformed to define its states according to the $\hat{\lambda}$ function (Equation 5). This entails unrolling traces through transient states, which may cause the size of the specification to grow exponentially.

Fortunately, in coherence protocols this is not the case. They exhibit a small upper bound on the number of intermediate transient states, because only a limited number of requests

from a peer must be handled before the node can conclude the processing of its own requests and reach the next stable state. In MESI, for example, the home node can only ask the remote to downgrade its state twice (from M/E to S, then from S to I). Furthermore, like other communication protocols, coherence protocols are sparse – only a few messages can be received from a valid peer implementation in any given state. Therefore, we expect the transformation to only yield a moderate increase in size.

V. APPLICATION TO A REAL IN-SILICON IMPLEMENTATION

While production hardware for the cache-coherent variants of protocol standards like CXL has yet to appear, we have applied our technique to the cache-coherent interconnect of the Enzian research computer [3]; indeed, this was a motivation for our original work. Enzian can be viewed as a 2-socket NUMA machine combining a Cavium ThunderX-1 48-core ARMv8 CPU with a large Field Programmable Gate Array (FPGA), which also implements the Cavium Coherent Processor Interconnect (CCPI), the CPU’s native inter-socket coherence protocol, appearing to the CPU as a second processor node.

While the ThunderX-1 was not originally intended to interoperate at the coherence level with anything other than another ThunderX-1, Enzian was designed to explore the space of emerging coherent heterogeneous platforms, and so must provide an endpoint implementation of CCPI on the FPGA that interoperates with the CPU. In the context of this work, we use the CPU’s Last-level cache (LLC) as the system under test, and use a combination of FPGA programming and software running on the CPU to exhaustively test the CPU’s silicon implementation of CCPI against our specification.

CCPI is a distributed directory-based MESI coherence protocol whose endpoints satisfy all of the requirements for semi-open-state testing: CCPI maintains cache line independence; it enforces sequential consistency and is able to cope with arbitrary reorderings of messages on the interconnect. Its endpoints further cleanly separate their input buffering and processing, and the design of their processing guarantees that a stable M-E-S-I protocol state (combination) is always reached after at most four transitions. As a result, CCPI’s behavior can be accurately described by two Mealy FSMs, one for a remotely-owned and one for a homed cache line, both of which return to a stable state in a bounded number of steps. Furthermore, software on the CPU can use hardware performance counters to determine if a cache line is in a transient or stable state, and for the latter, the state of each cache line can be explicitly read from the cache using privileged registers. Thus, the stable caching states are made visible in the protocol implementation, as required.

Our initial specification of the protocol endpoints was manually constructed from informal vendor documentation, and subsequently refined as a result of the testing process. The result is an exhaustively-tested specification of an in-silicon cache coherence protocol implementation.

A. Testing setup and methodology

Our testing setup operates on a single designated Cache Line Under Test (CLUT), and consists of three components: an *orchestrator* and a C library running on the CPU, and an FPGA testing component. The *orchestrator* is responsible for generating the test stimuli from the supplied specification and coordinating their execution. To generate test stimuli, the orchestrator performs the online graph traversal algorithm by Bourdonov et al. [1] while driving the implementation in tandem. The C library can issue operations (load, store, etc.) on the CLUT and can return the visible aspects of the CLUT’s current protocol state (transient or stable with a particular caching state) when invoked by the orchestrator. The FPGA tester implements the underlying reliable link protocol, can send and receive coherence messages to and from the CPU’s LLC when directed by the orchestrator, and relay received events back. Communication between the FPGA component and orchestrator cannot use the coherence protocol because it must not interfere with the CLUT state. We exploit the uncached I/O load/store operations that the ThunderX-1 supports as an out-of-band communication channel between the software orchestrator and the FPGA logic.

We must also prevent any other events in the system (such as conflict or capacity misses in the cache) from affecting the CLUT. We achieve this by placing the CLUT in a region of memory otherwise unused, and exploiting a feature of the CPU to “lock” it in the LLC, ensuring that the only operations that affect it are those explicitly initiated by the orchestrator.

This also addresses a further practical problem: since we are deliberately stalling the cache protocol, we run the risk of preventing the orchestrator itself making forward progress unless we can ensure that it does not need to initiate inter-node cache operations. In our current implementation, this can still occasionally happen due to global barrier operations we cannot control, deadlocking the interconnect and causing a “machine check” exception in the processor. In this case, we simply try again: the phenomenon does not affect the validity of a run that completes without a machine check.

B. Experience and results

Having developed our methodology and tools, and based on an incomplete understanding of CCPI derived from vendor documentation, it took approximately 2 person-days to formalize the behavior of the protocol in our specification format. It then took a further person-day to iteratively improve the specification based on testing feedback. A snippet from our specification is shown in Table I. It details some transition the home node can take when the remote node has the CLUT in shared.

The resulting remote node specification yields a successful verdict, establishing that the ThunderX-1’s CCPI remote node implementation is generalized-quasi-equivalent to this specification. This final specification has 107 transitions between the 4 stable M-E-S-I states and 42 additional transient states.

In an interoperability scenario, creating the specification would be done at most once against a “gold standard” ref-

s_K	t_U	x	\rightarrow	s'_K	t'_U	y	
I	S	[]	SI	\rightarrow	I	I []	\sim
I	S	[]	SE _d	\rightarrow	I	E []	AE
I	S	[]	LE	\rightarrow	I	S [(LE/IV)]	IV
I	S	[]	IE _d	\rightarrow	I	S [(IE _d /~)]	\sim

TABLE I: Excerpt from our home node specification: to the left of the arrow we denote the (state, input) pair consisting of last stable state s_K and the trace t_U since. To the right of the arrow we denote the state (s'_K, t'_U) the machine transitions to, and the output y generated. The symbol \sim denotes that no output is generated.

erence implementation, or would ideally be provided directly by the coherent interconnect standard.

Subsequent *conformance testing* against the specification is much quicker. Testing one transition of the ThunderX-1’s in-silicon implementation takes approximately 10 milliseconds, most of which is spent waiting between applying the input and observing the generated outputs to ensure that the ThunderX-1 has finished the processing of the former. Exhaustive testing of the remote node specification concludes in under two minutes, during which each stimulus is executed multiple times to exercise all potential non-deterministic behavior.

Moreover, our specification is human-readable and comprehensible, in part because it obviates the need for abstract state identifiers. With abstract identifiers, a reader would have to explicitly remember the context of each and every such identifier, something challenging for a protocol of this complexity. In contrast, our trace-based identifiers carry all the relevant context to determine at a glance which sequence of events has lead to that state, and eliminate the need to think about how to group behaviors into states, since every behavior has a unique representation in the specification.

We are convinced that the testing accurately reflects the implementation behavior. All the discrepancies revealed when iterating our manually-written specification were reasonable from a protocol perspective. We also uncovered behavior that we could not have known based on the documentation; for example, an undocumented error message that the remote node generates when it receives unexpected messages in some situations, or the elision of a particular protocol message in a way that correctly maintains coherence, yet is at odds with convention in the rest of the protocol implementation.

Finally, we observe completely deterministic behavior in testing. If our modelling had missed an important aspect of the protocol, we would expect this to manifest as non-deterministic behavior.

VI. CONCLUSION AND FUTURE WORK

Our experience shows that semi-open-state testing is a viable approach for testing state machine implementations. Building on existing open-state testing techniques allows us to design an efficient testing procedure, covering a significant space of real-world state machines. We evaluate these claims by testing the in-silicon state machine of a ThunderX-1 LLC

against a specification of a directory-based cache coherence protocol. Exhaustively testing the ThunderX-1 coherence implementation completes in a matter of minutes, demonstrating that our approach is also efficient in practice.

Our method can extend to specification *synthesis*, where we extract the behavior of a state machine implementation from an implementation. This is useful in situations where we observe interactions between implementations known to be good (by accident or design). Synthesis helps derive specifications where we only have access to implementations, e.g. when observing the interactions of two reference implementations.

The ThunderX-1 specification we test is restricted to two protocol actors. Proposed coherent interconnect standards like CXL allow many actors participating in a coherence protocol, and will require more sophisticated specifications to deal with the additional complexity. Testing whether the composition of heterogeneous cache coherence implementations provides cache coherence correctly is an important consideration for system integrators and hardware designers alike.

There are other real-world protocols that could benefit from semi-open-state testing, for example remote conformance of TCP stacks. Successful application of our approach in the context of other protocols and specifications would further demonstrate its general applicability.

Semi-open-state testing extends the set of state machine implementations that can be efficiently but exhaustively tested, to the case where a stable subset of implementation states is observable. We successfully apply semi-open-state testing to the in-silicon implementation of unmodified, real hardware.

VII. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and feedback, and we are grateful to Google for financial support.

REFERENCES

- [1] I. B. Bourdonov and A. S. Kossatchev, "Complete open-state testing of limitedly nondeterministic systems," *Programming and Computer Software*, vol. 35, no. 6, pp. 301–313, Nov. 2009. [Online]. Available: <https://doi.org/10.1134/S0361768809060012>
- [2] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, May 1978. [Online]. Available: <https://doi.org/10.1109/TSE.1978.231496>
- [3] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Feb. 2022, pp. 434–451. [Online]. Available: <https://doi.org/10.1145/3503222.3507742>
- [4] D. D. Sharma and I. Agarwal, "Compute Express Link 3.0 Standard," CXL Consortium, Tech. Rep., 2022.
- [5] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. [Online]. Available: <https://doi.org/10.1109/5.533956>
- [6] R. M. Hierons, "Testing from Partial Finite State Machines without Harmonised Traces," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1033–1043, Nov. 2017. [Online]. Available: <https://doi.org/10.1109/TSE.2017.2652457>

- [7] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 109–124. [Online]. Available: <https://doi.org/10.1145/186258.187153>
- [8] M.-C. Gaudel, "Software testing based on formal specification," in *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Springer Berlin Heidelberg, 2010, pp. 215–242. [Online]. Available: https://doi.org/10.1007/978-3-642-14335-9_7
- [9] R. M. Hierons, "FSM quasi-equivalence testing via reduction and observing absences," *Science of Computer Programming*, vol. 177, pp. 1–18, May 2019. [Online]. Available: <https://doi.org/10.1016/j.scico.2019.03.004>
- [10] F. Aarts, B. Jonsson, and J. Uijen, "Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction," in *Testing Software and Systems*, A. Petrenko, A. Simão, and J. C. Maldonado, Eds. Berlin, Heidelberg: Springer, 2010, pp. 188–204. [Online]. Available: https://doi.org/10.1007/978-3-642-16573-3_14
- [11] E. F. Moore *et al.*, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.
- [12] A. Dahbura, K. Sabnani, and M. Uyar, "Formal methods for generating protocol conformance test sequences," *Proceedings of the IEEE*, vol. 78, no. 8, pp. 1317–1326, Aug. 1990. [Online]. Available: <https://doi.org/10.1109/5.58319>
- [13] D. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413–426, Apr. 1989. [Online]. Available: <https://doi.org/10.1109/32.16602>
- [14] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic State Machines in Protocol Conformance Testing," in *Proceedings of the IFIP TC6/WG6. 1 Sixth International Workshop on Protocol Test systems VI*, Jan. 1993, pp. 363–378.
- [15] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1998.
- [16] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Computing Surveys*, vol. 29, no. 1, pp. 82–126, Mar. 1997. [Online]. Available: <https://doi.org/10.1145/248621.248624>
- [17] —, "Formal verification of complex coherence protocols using symbolic state models," *Journal of the ACM*, vol. 45, no. 4, pp. 557–587, Jul. 1998. [Online]. Available: <https://doi.org/10.1145/285055.285057>
- [18] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 247–260. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00030>
- [19] H. Kahlouche, C. Viho, and M. Zendri, "Hardware Testing Using a Communication Protocol Conformance Testing Tool," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer, 1999, pp. 315–329. [Online]. Available: https://doi.org/10.1007/3-540-49059-0_22
- [20] A. Kriouile and W. Serwe, "Using a formal model to improve verification of a cache-coherent system-on-chip," in *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, 2015, pp. 708–722.
- [21] J. You, D. Bural, J. Brown, and J. Zbiciak, "Red Baron: Near/post-silicon SoC cache coherence stress tester," in *2016 IEEE Dallas Circuits and Systems Conference (DCAS)*, Oct. 2016, pp. 1–4. [Online]. Available: <https://doi.org/10.1109/DCAS.2016.7791135>
- [22] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *2008 IEEE International Conference on Computer Design*, Oct. 2008, pp. 348–355. [Online]. Available: <https://doi.org/10.1109/ICCD.2008.4751884>
- [23] D. Foley and J. Danskin, "Ultra-Performance Pascal GPU and NVLink Interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.
- [24] CCIX Consortium and others, "Cache Coherent Interconnect for Accelerators (CCIX)," January 2019. [Online]. Available: <http://www.ccixconsortium.com>

- [25] W. W. Terpstra, “TileLink: A free and open-source, high-performance scalable cache-coherent fabric designed for RISC-V,” in *Proc. 7th RISC-V Workshop*, 2017.
- [26] M. Martin, “Formal verification and its impact on the snooping versus directory protocol debate,” in *2005 International Conference on Computer Design*, Oct. 2005, pp. 543–549. [Online]. Available: <https://doi.org/10.1109/ICCD.2005.58>