Automated Conversion of Axiomatic to Operational Models: Theory and Practice

Adwait Godbole∗†‡, Yatin A. Manerkar†, and Sanjit A. Seshia‡△

‡University of California Berkeley, Berkeley, USA  △University of Michigan, Ann Arbor, USA

Abstract—A system may be modelled as an operational model (which has explicit notions of state and transitions between states) or an axiomatic model (which is specified entirely as a set of invariants). Most formal methods (e.g., IC3, invariant synthesis, etc) are designed for operational models and are largely inaccessible to axiomatic models. Furthermore, no prior method exists to automatically convert axiomatic models to operational ones, so operational equivalents to axiomatic models had to be manually created and proven equivalent.

In this paper, we advance the state-of-the-art in axiomatic to operational model conversion. We show that general axioms in the µspec axiomatic modelling framework cannot be translated to equivalent finite-state operational models. We also derive restrictions on the space of µspec axioms that enable the feasible generation of equivalent finite-state operational models for them. As for practical results, we develop a methodology for automatically translating µspec axioms to equivalent finite-state automata-based operational models. We demonstrate the efficacy of our method by using the models generated by our procedure to prove the correctness of ordering properties on three register-transfer-level (RTL) designs.

I. INTRODUCTION

When modelling hardware or software systems using formal methods, one traditionally uses operational models (e.g. Kripke structures [1]), which have explicit notions of state and transitions. However, one may also model a system axiomatically, where instead of a state-transition relation, the system is specified entirely by a set of axioms (e.g., invariants) that it maintains. Executions that obey the axioms are allowed, and those that violate one or more axioms are forbidden. The vast majority of formal methods works use the operational modelling style. However, axiomatic models have been used to great effect in certain domains such as memory models, where they have shown order-of-magnitude improvements in verification performance over equivalent operational models [2].

Operational and axiomatic models each have their own advantages and disadvantages [3]. Operational models can be more intuitive as they typically resemble the system that they are modelling. Hence one is not required to reason about invariants to write the model. On the other hand, axiomatic models tend to be more concise and potentially offer faster verification [2].

Many formal methods (e.g., refinement procedures [4], invariant synthesis, IC3/PDR [5], [6]) are set up to use operational models. Axiomatic models are largely or completely incompatible with these techniques, as the axioms constrain full traces rather than a step of the transition relation. One way to take advantage of these techniques when using axiomatic models is to create and use operational models equivalent to the axiomatic models. The only prior method of doing this was to first manually create the operational model and then manually prove it equivalent to the axiomatic model. There have been several works doing so [2], [7], [8], [9], [10].

Manually creating an operational model and proving equivalence is cumbersome and error-prone. The ability to automatically generate operational models equivalent to a given axiomatic model would be beneficial, eliminating both the time spent creating the operational model as well as the need for tedious manual equivalence proofs. Generated models can then be fed into techniques currently requiring operational models (e.g. IC3/PDR).

To this end, we make advances in this paper towards the automatic conversion of axiomatic to equivalent operational models, on both theoretical and practical fronts. In our work, we focus specifically on µspec [11], a well-known axiomatic framework for modelling microarchitectural orderings, which has been used in a wide range of contexts [12], [13], [14], [15], [16] including memory consistency, cache coherence and hardware security.

On the theoretical front, we show that it is impossible to convert general µspec axioms to equivalent finite-state operational models. However, we show that it is feasible to generate equivalent operational models for a specific subset of µspec (henceforth referred to as µspecRE). On the practical side, we develop a method to automatically translate universal axioms\(^1\) in µspecRE into equivalent finite-state operational models comprised of building blocks we term as axiom automata (finite automata that monitor whether an axiom has been violated). Furthermore, for arbitrary µspec axioms, our method can generate operational models that are equivalent to the axioms up to a program-size bound.

To evaluate our technique, we convert axioms for three RTL designs to their corresponding operational models: an in-order multicore processor (multi_vscale), a memory-controller (sdram_crtl), and an out-of-order single-core processor (tomusulo). We showcase how the generated models can be used with procedures like BMC and IC3/PDR which are usually inaccessible for axiomatic models, and we produce both bounded and unbounded proofs of correctness.

Overall, the contributions of this work are as follows:

- We prove that generation of equivalent finite-state operational models for arbitrary µspec axioms is impossible.

\(^1\)Axioms that do not contain \(\exists\) quantifiers.
We provide a procedure for generating equivalent finite-state operational models for universal axioms in µspecRE.

We propose the axiom-automata formulation to generate equivalent finite-state operational models from universal axioms in µspecRE (or from arbitrary µspec axioms if only guaranteeing equivalence up to a bounded program size).

We evaluate our method for operational model generation by using our generated models to prove the (bounded/unbounded) correctness of ordering properties on three RTL designs: multi_vsacle, tomasulo, and sdram_ctrl.

**Generality.** While axiomatic models enforce constraints over complete executions, operational models do this local to each transition. Ensuring that behaviours generated by the latter are also allowed by the former requires performing non-local consistency checks which are hard to reason about, especially for unbounded executions. This has been observed in manual operationalization works as well. Taking the example of [7], (which operationalizes C11), we address issues of eliminating consistent executions too early [7, §3] and repeatedly checking consistency [7, §4] by developing concepts such as t-reordering boundedness (Def. 6) and extensibility (Def. 7). Though we focus on µspec, we believe many of the underlying challenges and concepts carry over to frameworks such as Cat [2].

**Outline.** §II covers the syntax and semantics of µspec used in this paper. §III covers the formulation of the space of operational models we consider. They have finite control-state and read-only input tapes for the instruction streams (programs) executed by each core. §IV defines our notions of soundness, completeness, and equivalence when comparing operational and axiomatic models. In §V, we show that it is impossible to synthesise equivalent finite-state operational models from arbitrary axiomatic models. We develop an underapproximation, called t-reordering boundedness, that addresses this by bounding the depth of reorderings possible. In §VI we restrict µspec further by requiring extensibility (preventing current events from influencing orderings between previous events). Restricting µspec by t-reordering boundedness and extensibility is sufficient to enable the automatic generation of equivalent finite-state operational models (Thm. 2). §VII describes our conversion procedure based on axiom automata. §VIII evaluates our technique by using it to generate operational models, which are then used for checking properties of RTL designs. §IX covers related work, and §X concludes, with §XI suggesting avenues for future work. This paper is accompanied by an extended version which contains supplementary material and proofs [17].

### II. µspec Syntax and Semantics

#### A. µspec Syntax

\[
\langle AX \rangle := \forall i \ AX \mid \exists i \ AX \mid \phi(i_1, \ldots, i_m)
\]

\[
\langle \phi \rangle := \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid (\text{atom})
\]

\[
\langle \text{atom} \rangle := i_1 <_r i_2 \mid \text{hb}(i_{1,2}, i_{3,4}) \mid P(i_1, \ldots)
\]

\[
\langle \text{st} \rangle := \text{Fet} \mid \text{Dec} \mid \text{Exe} \mid \text{WB} \mid \cdots
\]

Fig. 2: µspec Syntax.
ax0: \forall i1. \text{hb} (i1.\text{Exe}, i1.\text{Com})
ax1: \forall i1, i2. (i1 < i2 \land \text{DepOn} (i1, i2)) 
\implies \text{hb} (i1.\text{Exe}, i2.\text{Exe})
ax2: \forall i1, i2. \text{SameCore} (i1, i2) \implies 
(\text{hb} (i1.\text{Exe}, i2.\text{Exe}) \lor \text{hb} (i2.\text{Exe}, i1.\text{Exe}))
ax3: \forall i1, i2. i1 < i2 \implies \text{hb} (i1.\text{Com}, i2.\text{Com})

Fig. 3: An example axiomatic model.

B. Illustrative \mu spec Example

Consider the four axioms in Fig. 3. In the axioms, \( i1, i2 \) are instruction variables and \( \text{Exe, Com} \) are stage names (short for execute and commit respectively). The axiom \( ax0 \) requires that for each instruction, the execute stage (\text{Exe}) of that instruction must happen before the commit stage (\text{Com}). Intuitively, \( ax1 \) says that when \( i2 \) depends on \( i1 \) (captured by the predicate \text{DepOn}), \( i1 \) should be executed before \( i2 \); \( ax2 \) says that the execute events of instructions on the same core should be totally ordered by \text{hb}. The third axiom \( ax3 \) says that when \( i1 \) and \( i2 \) are in program order (denoted by \( < \)), \( i1 \) must be committed before \( i2 \).

Fig. 4 shows valid and invalid execution graphs for the program snippet in Fig. 5. The snippet is of a 2-core program, with two instructions per core. Instruction \( i1 \) is dependent on the result of \( i0 \) (since its source register is the same as the destination of \( i0 \)). In the example axiomatic semantics, \( ax1 \) requires that the execute event of instruction \( i0 \) be before that of \( i1 \). The execution in Fig. 4b is invalid w.r.t. \( ax1 \) since \( i1.\text{Exe} \) is executed before \( i0.\text{Exe} \). The execution in Fig. 4a is valid even though the \( i2.\text{Exe} \) and \( i3.\text{Exe} \) events are reordered since \( i3 \) does not depend on \( i2 \). Both executions are valid w.r.t. \( ax0, ax2 \) and \( ax3 \).

C. Programming Model

We consider multi-core systems with each core executing a straight-line program over a finite domain of operations. This is common in memory models [2], [12], [16], [20] and distributed systems [21] literature.

1) Cores: The system consists of \( n \) processor cores: \( \text{Cores} = [n] \). Each core executes operations from a finite set \( \mathbb{O} \). The axiomatic model \( A \) assigns predicates from \( P \) an interpretation over the universe \( \mathbb{O} \). We denote this interpretation as \( P^A \subseteq O^A \) for an arity-\( k \) predicate.

2) Instruction streams: An instruction stream \( I \) is a word over \( \mathbb{O}: I \in \mathbb{O}^* \). A program \( P \) is a set of per-core instruction streams: \( \{I_c\}_{c \in \text{Cores}} \). For a core \( c \) and label \( 0 \leq j < |I_c| \), we call the triple \( (c,j,I_c[j]) \) an instruction\(^2\). We denote components of instruction \( i = (c,j,I_c[j]) \), as: \( c(i) = c, \) label \( \lambda(i) = j \) and operation \( op(i) = I_c[j] \). The set of instructions occurring in \( P \) is: \( \text{instrsOf}(P) = \{(c,j,I_c[j]) \mid c \in \text{Cores}, 0 \leq j < |I_c|\} \) and the set of all possible instructions as \( I = \text{Cores} \times \mathbb{Z}^+ \times \mathbb{O} \).

3) Instruction stages: Instruction execution in \( \mu spec \) is decomposed into stages. The set of stages, \( \text{Stages} \), is a parameter of the semantics. Instruction \( i \) performing in stage \( st \), (i.e. \( i.st \)) is an atomic event in an execution. The execution of \( P \) is composed of the set of events: \( \text{eventsOf}(P) = \{i.st \mid i \in \text{instrsOf}(P), st \in \text{Stages}\} \). The set of all possible events is \( E = \{i.st \mid i \in I, st \in \text{Stages}\} \).

Definition 1 (Event). An event \( e \) is of the form \( i.st \). It represents the instruction \( i \in I \), (atomically) performing in stage \( st \in \text{Stages} \).

Example 1. Following the example in Fig. 5 we consider an architecture with two opcodes: \text{add}, \text{lw} for add and load respectively. For each of these, we may have several actual operations (with different operands), thus giving us the set \( \mathbb{O} \). The program \( P \) in Fig. 5 has two cores: \( \text{Cores} = \{c0, c1\} \) and four instructions: \( \text{instrsOf}(P) = \{i0, i1, i2, i3\} \). We have, for example, \( c(i1) = c0, \lambda(i1) = 1, op(i1) = \text{add} \), \( r3, r2, r1 \) while \( c(i2) = c1, \lambda(i2) = 0 \). The instruction stream for core \( c0 \) is \( I_0 = i0 \cdot i1 \land \text{that of core } c1 \) is \( I_1 = i2 \cdot i3 \).

Let us suppose that this program is executed on a 4-stage microarchitecture with \( \text{Stages} = [\text{Fet, Dec, Exe, WB, Com}] \). The events corresponding to the program are given by \( \text{eventsOf}(P) = \{i0.\text{Fet}, i0.\text{Dec}, \cdots, i3.\text{Com}\} \) with \( |\text{eventsOf}(P)| = 4 \times 5 = 20 \).

D. Formal \mu spec Semantics

We now define the formal semantics of \( \mu spec \) axioms.

Definition 2 (\( \mu hb \) graph). For a program \( P \), a \( \mu hb \) graph is a directed acyclic graph, \( G(V,E) \), with nodes \( V = \text{eventsOf}(P) \) representing events and edges representing the happens-before relationships, i.e. \( (e1, e2) \in E \equiv \text{hb}(e1, e2) \).

Validity of \( \mu hb \) graph w.r.t. an axiomatic semantics: Consider an axiomatic semantics \( \mathcal{A} \) (i.e. a set of axioms). A \( \mu hb \) graph \( G = (V,E) \) is said to represent a valid execution of program \( P \) under \( \mathcal{A} \) if it satisfies all the axioms in \( \mathcal{A} \). We denote the validity of a \( \mu hb \) graph \( G \) by \( G \models \mathcal{A} \).

Satisfaction w.r.t. an axiom: We first define satisfaction for the quantifier-free part, starting at the atoms. Let \( s: I(AX) \to I \) be an assignment for the symbolic instruction variables \( I(AX) \) in axiom \( AX \).

\[
\begin{align*}
G \models 1s &< r_i 2s \iff c(s(1i)) = c(s(1i)) \land \\
&\lambda(s(1i)) < \lambda(s(1i)) \quad \cdots (i) \\
G \models P(i_1, \cdots, i_n)[s] &\iff \\
&\text{op}(s(1i)), \cdots, \text{op}(s(1i)) \in P^A \quad \cdots (ii) \\
G \models \text{hb}(i_1.st_1, i_2.st_2)[s] &\iff \\
&(s(1i)).st_1, s(1i).st_2 \in E^+ \quad \cdots (iii)
\end{align*}
\]

In (i), the reference order \( <r_i \) relates instructions \( i_1, i_2 \) from the same instruction stream if \( i_1 \) is before \( i_2 \). In (ii) we extend predicate interpretations, \( P^A \), (defined over \( \mathbb{O} \)) to instructions by taking the \( \text{op}(-) \) component. Finally, \( \text{hb} \) atoms
are interpreted as $E^+$, i.e. transitive closure of $E$, as stated in (iii). Operators $\land, \lor, \neg$ have their usual semantics.

We now define the satisfaction of a (quantified) axiom $AX$ by a graph $G$, denoted by $G \models p \ AX$ above.

$G \models p \phi[s] \iff G \models \phi[s]$

for quantifier-free $\phi$.

$G \models p \ \forall i \ \phi[s_i] \iff G \models p \ \phi[s[i \leftarrow i]]$

for all $i \in \text{instrsOf}(P) \setminus \text{range}(s)$.

$G \models p \ \exists i \ \phi[s_i] \iff G \models p \ \phi[s[i \leftarrow i]]$

for some $i \in \text{instrsOf}(P) \setminus \text{range}(s)$.

The base case is $G \models p \phi[s]$ (where $\phi$ is quantifier-free) and follows the earlier definitions. We extend $G \models p \phi$ with (almost) usual quantification semantics: $\forall (\exists)$ quantifies over all (some) instructions in $\text{instrsOf}(P)$. Execution $G$ is a valid execution of $P$ under semantics $A$, denoted as $G \models p \ AX$, if $G \models p \ AX$ for all axioms $AX$ in $A$.

III. OPERATIONAL MODEL OF COMPUTATION

To concretize our claims, we introduce a model of computation that characterizes the models of interest. We choose to focus on finite-state operational models that generate totally ordered traces, where transitions represent (i/st) events. While there are less restrictive models (e.g. event structures [22], [23]), such models require specialized, typically underapproximate, verification techniques (e.g. [24], [25], [26]).

Our choice is motivated by the ability to (a) have finite-state implementations of generated models (e.g. in RTL) and (b) verify against these models with off-the-shelf tools (e.g. model checkers using BDD and SMT-based backends).

A. Model of computation

Intuitively, the model of computation resembles a 1-way transducer [27], [28] with multiple (read-only) input tapes (one tape for each instruction stream). This allows us to execute programs of unbounded length with a finite control state. 3

1) Model definition: An operational model is parameterized by cores $\text{Cores}$, stages $\text{Stages}$, and a history parameter $h \in \mathbb{N} \cup \{\infty\}$ which bounds the length of tape to the left of the head. It is a tuple $(Q, \Delta, q_{init}, q_{final})$:

- $Q$ is a finite set of control states
- $\Delta$ is a finite set of control instructions
- $q_{init} \in Q$ is the initial state
- $q_{final} \in Q$ is the final state which must be absorbing (i.e. it has a self-loop)

A model is finite-state if $Q$ is finite, and it has bounded-history if $h \in \mathbb{N}$. For the end goal of effective verification, we are interested in finite-state, bounded-history models since it is precisely such models that can be compiled to finite-state systems.

2) Model semantics: A configuration is a triple $\gamma = (U, q, V)$ where $U : \text{Cores} \rightarrow \mathbb{I}^*, V : \text{Cores} \rightarrow \mathbb{I}^*$ and $q \in Q$. Intuitively $U(V)$ represent, for each instruction stream, the contents of the input tape to the left (right) of the head respectively. For a bounded history machine, a configuration is allowed only if $|U(c)| \leq h$ for all $c \in \text{Cores}$. For unbounded history all configurations are allowed.

The set of actions is

$\text{Act} = \{\text{right}(c) \mid c \in \text{Cores}\} \cup \{\text{stay}\} \cup \{\text{sched}(c, i, s) \mid c \in \text{Cores}, s \in \text{Stages}, i \in [h]\} \cup \{\text{drop}(c, i) \mid c \in \text{Cores}, i \in [h]\}$

Intuitively, these represent in order: motion of the tape head for $c$ to the right, silent (no-effect), generation of an event, and removing the $i^{th}$ instruction from the left of the head. We provide full semantics in the supplementary material [17].

For word $w \in \mathbb{I}^*$, let $\text{fst}(w)$ denote its first element if $w \neq \epsilon$ and $\text{head}$ otherwise. Transitions are enabled based on the control state and the instructions that the tape-heads point to: transition $(q_1, (i_1, \cdots, i_{|\text{Cores}|}), q_{2,\infty}) \in \Delta$ is enabled in configuration $\gamma = (U, q, V)$ if $q_1 = q$ and $\text{fst}(V(c)) = i_c$ for each $c \in \text{Cores}$.

3) Runs: The initial configuration is given by $\gamma_{init}(P) = (U_{init}, q_{init}, V_{init})$ where $U_{init} = \lambda c, \epsilon, V_{init} = \lambda c, c \in \text{I}_c$, i.e. for each core, the left of the tape head is empty, and the right of the tape head consists of the instruction stream for that core. Starting from $\gamma_{init}(P)$, the machine transitions according to the transition rules. Such a sequence of configurations $\gamma_{init}(P) = \gamma_0 \overset{e_1}{\rightarrow} \gamma_1 \cdots \overset{e_m}{\rightarrow} \gamma_m$, where all $\gamma_i$ are allowed is called a run. A run is called accepting if it ends in the state $q_{final}$.

4) Traces: The sequence of event labels $\sigma = e_1 \cdots e_m$ annotating a run is the trace corresponding to the run. Each label is an event from $E$ and hence $\sigma \in E^*$. We view $\sigma$ as a (linear) $\mu\text{hb}$ execution graph $e_1 \overset{\text{hb}}{\rightarrow} e_2 \cdots \overset{\text{hb}}{\rightarrow} e_m$, and hence define $\sigma \models A$ in the usual way. Accordingly, we will sometimes refer to $\sigma$ as an execution of a program $P$. The set

![Fig. 4: Valid (a) and an invalid (b) execution graphs for the program in Fig. 5 and axioms in Fig. 3. All edges represent the $\text{hb}$ relation. The red (bold) edge violates ax1.](image-url)
of traces corresponding to accepting runs of an operational model $M$ on a program $P$ are denoted as $\text{traces}_M(P) \subseteq E^+$. 

IV. SOUNDNESS, COMPLETENESS, AND EQUIVALENCE

We proceed to formalize the notion of equivalence that relates axiomatic and operational models. In literature [29], [2], ISA-level behaviours of programs have been annotated by the read values of load operations. Hence, one notion of equivalence might be to require that identical read values be possible between the models. While this may be reasonable for ISA-level behaviours, it can hide microarchitectural features: different microarchitectural executions can have identical architectural results. Given that $\mu$spec models executions at the granularity of microarchitectural events, we adopt a stronger notion of equivalence. For soundness, we require that the operational semantics generates linearizations of $\mu_{hb}$ graphs that are valid under the axiomatic semantics. Formally:

**Definition 4 (Soundness).** An operational model $M$ is sound w.r.t. $A$ if for any program $P$, each trace in $\text{traces}_M(P)$ is a linearization of some $\mu_{hb}$ graph that is valid under $A$.

Before defining completeness, we need to address a subtlety. Since operational executions are viewed as $\mu_{hb}$ graphs by interpreting trace-ordering as the $\text{hb}$ ordering, the operational model always generates linearized $\mu_{hb}$ graphs. However, in general, linearizations of valid $\mu_{hb}$ graphs could end up being invalid w.r.t the axioms. Consider Example 2.

**Example 2 (Non-refinable axiom).** For the following axiom with Stages $= \{S\}$, the graph (a) is a valid execution. However, both of its linearizations (b) and (c) are invalid. Thus, $\mu$ becomes invalid w.r.t the axioms. Consider Example 2.

To address this issue, we develop the notion of refinability. For two $\mu_{hb}$ graphs $G = (V, E)$ and $G' = (V', E')$, we say that $G'$ refines $G$, denoted $G \subseteq G'$ if (1) $V = V'$ and (2) $(e_1, e_2) \in E^+ \implies (e_1, e_2) \in E'^+$.

**Definition 4 (Refinable $\text{hb}$).** An axiomatic semantics $A$ is refinable if for any program $P$, and $\mu_{hb}$ graph $G$ s.t. $G \models_P A$, we have $G' \models_P A$ for all linear graphs $G'$ satisfying $G \subseteq G'$.

Refinability says that all linearizations of a valid graph are valid. While executions under axiomatic semantics are given by (partially-ordered) $\mu_{hb}$ graphs, our class of operational models generate totally-ordered traces. Refinability bridges this gap by relating valid $\mu_{hb}$ graphs to valid traces. Interestingly, we can check whether a universal axiomatic semantics satisfies refinability, which at a high level, we show via a small model property (Lemma 1).

**Lemma 1.** Given a universal axiomatic semantics we can decide whether the semantics is refinable.

Refinability is especially important for completeness. For non-refinable semantics, validity of linearizations cannot be checked based on the axioms, as all linearizations may be invalid (Example 2).

We assume that the axiomatic semantics satisfies refinability.

We define completeness and our formal problem statement.

**Definition 5 (Completeness).** An operational model $M$ is complete, if for any program $P$ and valid $\mu_{hb}$ graph $G \models_P A$, $\text{traces}_M(P)$ contains all linearizations of $G$.

**Formal Problem Statement** Given an axiomatic semantics $A$, a set of cores $\text{Cores}$ and stages $\text{Stages}$, generate a finite state, bounded history model, $M = (Q, \delta, q_{init}, q_{final})$, which satisfies soundness and completeness (Defns. 3 and 5).
can be ordered arbitrarily, while \( T \) stage events must maintain the same ordering as that of corresponding \( S \) stages. Hence the machine must remember the \( S \) orderings in its finite control. However, the number of such orderings grows (exponentially) with the number of instructions \( m \), implying that existence of a finite-state model that works for all programs is not possible.

**Corollary 1.** There does not exist a finite state operational model (even with \( h = \infty \)) which is sound and complete with respect to the \( A^\# \) axioms.

### B. An underapproximation result

Given the results of the previous section, we must relax some constraint imposed on the operationalization: we choose to relax completeness. To do so, we define an underapproximation called \( t \)-reordering bounded traces. Intuitively, this imposes two constraints: (a) it bounds the depth of reorderings between instructions on each core, (b) it bounds the number of instructions executed on all other cores, while a core is executing a single instruction.

We observe that (a) is a reasonable assumption since most microarchitectures bound reordering depth, often due to finite reorder buffers. On the other hand, (b) can be thought of as a fairness/starvation-freedom property.

For two instructions \( i_1, i_2 \) on the same core, let \( \text{diff}_r(i_1, i_2) = \lambda(i_2) - \lambda(i_1) \) (recall that \( \lambda(i) \) is the instruction index of \( i \)). Consider a trace \( \sigma \) of program \( P \). For \( i \in \text{instrsOf}(\sigma) \), we define the starting index of \( i \), denoted as \( \text{start}(i) \), as the index of the first event of instruction \( i \) in \( \sigma \). Similarly we define the ending index, \( \text{end}(i) \) as the largest index for some event of \( i \) in \( \sigma \). Let the prefix-closed end index of \( i \) be the max of end over instructions that are \( \leq_r i \): \( \text{pfxend}(i) = \max\{\text{end}(i') \mid i' \leq_r i\} \). Two instructions \( i_1 \) and \( i_2 \) are coupled in a trace (denoted as \( \text{coup}(i_1, i_2) \)) if the intervals \( [\text{start}(i_1), \text{pfxend}(i_1)], [\text{start}(i_2), \text{pfxend}(i_2)] \) overlap.

**Definition 6** (\( t \)-reordering bounded traces). A trace is \( t \)-reordering bounded if, for any pair of instructions \( i_1, i_2 \) with \( \text{c}(i_1) = \text{c}(i_2) \), (1) if \( i_2, \text{st}_2 \xrightarrow{\text{hb}} i_1, \text{st}_1 \) then \( \text{diff}_r(i_1, i_2) < t \) and (2) if \( \text{coup}(i_1, i_2) \), \( \text{coup}(i_1, i_2) \) for some \( i \) then \( |\text{diff}_r(i_1, i_2)| < t \).

Intuitively, (1) says that an instruction cannot be reordered with another that precedes it by \( t \) indices, while (2) says that instructions on a core cannot be \textit{stalled} while more than \( t \) instructions are executed on another. Note that \( t \)-reordering boundedness is a property of traces, and not of axioms. We now relax completeness (and hence equivalence) to require that the operational model at least generate all \( t \)-reordering bounded linearizations (instead of all linearizations).

**Definition 5\# (\( t \)-completeness).** An operational model \( M \) is \( t \)-complete w.r.t. an axiomatic model \( A \), if for each program \( P \) and \( G \models P \text{ A} \), traces\(_M\)(\( P \)) contains all \( t \)-reordering bounded linearizations of \( G \).

Replacing Defn. 5 with its \( t \)-bounded relaxation (Defn. 5\#) addresses the issue of having to keep track of an unbounded number of orderings. However, to allow for finite implement-
In Fig. 7, for example, the first instruction stream of \( P \) is \( i_0 \cdot i_1 \cdot i_3 \cdot i_4 \). The prefix program \( P' \) has (the prefix) \( i_0 \cdot i_1 \cdot i_2 \) as its first instr. stream. On the other hand, the residual program, \( P'' = P \otimes P' \), has the suffix \( i_3 \) as its instruction stream.

For graphs \( G' = (V', E') \) and \( G'' = (V'', E'') \), with \( V' \cap V'' = \emptyset \) we define \( G' \triangleright G'' \) as the graph \( G = (V, E) \) where, (1) \( V = V' \cup V'' \), and (2) \( E = E' \cup E'' \cup \{(e', e'') \mid e' \in \text{sink}(E'), e'' \in \text{source}(E'')\} \). The example in Fig. 7 illustrates such a composition: we have \( G = G' \triangleright G'' \).

**Definition 7** (Extensibility). An axiom \( AX \) satisfies extensibility if for any programs \( P \) and \( P' \) s.t. \( P' \preceq P \), and \( P'' = P \otimes P' \) if \( G' \models P \) and \( G'' \models P' \), then \( G'' \models P' \). An axiomatic semantics \( A \) satisfies extensibility if all axioms \( AX \in A \) satisfy extensibility.

We require that the axiomatic model satisfies extensibility. We define \( \mu \specRE \) (RE stands for Refinable, Extensible) as the subset of \( \mu \spec \) in which all axioms are refinable and extensible. Finite-state, bounded-history synthesis is feasible for universal axioms in \( \mu \specRE \), as we discuss in the next section. Like refinability, we can check whether an axiom satisfies extensibility (Lemma 2).

**Lemma 2.** Given a universal axiom we can decide whether it satisfies extensibility.

### VII. CONVERTING TO OPERATIONAL MODELS USING AXIOM AUTOMATA

In this section, we describe our approach that converts an axiomatic model into an equivalent operational model \( \mathcal{M} \). In §VII-A we develop axiom automata, which are the building blocks of our operationalization: they are automata that check for axiom compliance as the operational model executes. In §VII-B we describe how these automata can be instantiated to ensure validity for bounded programs with arbitrary \( \mu \spec \) axioms. §VII-C holds our main result: we describe how axiom automata can be instantiated to get a finite-state bounded-axioms. §VII-C holds our main result: we describe how axiom automata can be instantiated to get a finite-state bounded-axioms. §VII-C holds our main result: we describe how axiom automata can be instantiated to get a finite-state bounded-axioms.

**A. Axiom Automata**

In what follows, we fix a (universal) axiom \( AX = \forall i_1 \cdots i_k \cdot \phi(i_1, \ldots, i_k) \), and let \( I(AX) = \{i_1, \ldots, i_k\} \). \( E(AX) = \{i.\text{st} \mid i \in I(AX), \text{st} \in \text{Stages}\} \). This axiom enforces that \( \phi(\cdot) \) holds for all \( k \)-tuples of instructions in the given program. An axiom automaton is a finite state automaton that monitors whether \( \phi(\cdot) \) holds for a single \( k \)-tuple of instructions. Our operational model is composed of several such automata - thereby allowing us to check all \( k \)-tuples. We now define axiom automata, starting with some auxiliary definitions.

Let \( \text{nonhb}(AX) \) denote the non-\( \text{hb} \) atoms in \( \phi \), i.e. instruction predicate applications and \( <_c \) orderings. A context is an assignment (of true/false) to each atom in \( \text{nonhb}(AX); \mu \spec : \text{nonhb}(AX) \rightarrow \mathbb{B} \). Each variable assignment \( s : I(AX) \rightarrow \mathbb{I} \) fixes the valuation of all \( \text{nonhb}(AX) \) atoms (following the semantics in §II). Hence each assignment \( s \) leads to a unique context, which we denote as \( \mu \spec(s) \).

We extend assignments to events and words over events. For \( e = i.\text{st} \), we define \( \mu \spec(e) = s(i).\text{st} \) and for \( w \in E(AX)^* \),

\[ s(w) = s(w[0]) \cdots s(w[w - 1]) \in E^\ast \]

As mentioned in §III-A4, we interpret \( s(w) \in E^\ast \) as the \( \mu \text{hb} \) graph \( w[0] \rightarrow w[1] \cdots \rightarrow w[w - 1] \).

Observe that once we fix the context, the validity of \( \phi(\cdot) \) only depends on the value of the \( \text{hb} \) atoms in \( \phi \). Hence for two assignments \( s_1, s_2 \) with the same context: \( \mu \spec(s_1) = \mu \spec(s_2) \), \( s_1 \) and \( s_2 \) share the same set of valid executions: \( s_1(w) \) satisfies \( \phi(\cdot) \) only if and only if \( s_2(w) \) does. This implies that across different assignments \( s \), there are only finitely many valid sets of executions over events in \( E(AX) \) - one for each context. Intuitively, contexts divide the set of all possible assignments into classes which admit similar orderings.

As a consequence of the above, for each \( AX \) and context \( \mu \spec \), we can construct a finite state automaton that recognizes acceptable orderings of \( E(AX) \) (Lemma 3). The main observation behind Lemma 3 is that once the context (i.e. interpretation of the \( \text{nonhb}(AX) \) atoms) is fixed, the allowed orderings can be represented as a language over the symbolic events \( E(AX) \).}

**Lemma 3 (Axiom-Automata).** Given an axiom \( AX \) and context \( \mu \spec \), there exists a finite-state automaton \( \text{aa}(AX[\mu \spec]) \) over alphabet \( E(AX) \) with language \( \{w \mid w \in E(AX)^\ast, s(w) \models \phi(\cdot)\} \) for all \( s \) that agree with \( \mu \spec \).

**B. Deploying axiom automata**

1) Concretization of an axiom automaton: The automaton \( \text{aa}(AX[\mu \spec]) \) mentioned in Lemma 3 recognizes orderings over the symbolic alphabet \( E(AX) \) that lead to \( \phi(\cdot) \) being satisfied. Our end goal, however, is identifying acceptable orderings over the (non-symbolic) events \( E \). This requires us to generate concrete instances of axiom automata, one for each assignment \( s : I(AX) \rightarrow \mathbb{I} \), which we now do.

Given an assignment \( s : I(AX) \rightarrow \mathbb{I} \), we denote the (concretized) automaton for \( s \) w.r.t \( AX \) as \( \text{aa}(AX, s) \). The automaton \( \text{aa}(AX, s) \) is identical to \( \text{aa}(AX[\mu \spec(s)]) \), except that the symbolic alphabet \( E(AX) \) replaced by its image \( s(E(AX)) \) under \( s \). Intuitively (by §VII-A), the set of valid orderings of events in \( s(E(AX)) \) is characterized by the context of \( s \), \( \mu \spec(s) \). This means that the acceptable orderings of events in \( s(E(AX)) \) is identical to the set of words (orderings) accepted by \( \text{aa}(AX[\mu \spec(s)]) \), except that the symbolic events \( E(AX) \) should be replaced by their concrete counterparts, \( s(E(AX)) \). This justifies the definition of \( \text{aa}(AX, s) \).

We extend the notation \( \text{aa}(AX, s) \) from a single assignment to a set of assignments. For \( I \subseteq \mathbb{I} \), we denote by \( \text{aa}(AX, I) \) the set of axiom automata over \( I : \{\text{aa}(AX, s) \mid s : I(AX) \rightarrow I\} \).

2) A basic operationalization: Lemma 3 and the concretization defined in §VII-B1 suggest an operationalization for \( AX \). For a program \( P \), if a trace \( \sigma \) is accepted by all (concrete) automata \( \text{aa}(AX, \text{instsOf}(P)) \) then \( \sigma \models \phi[s] \) holds for each assignment \( s \), thus satisfying \( AX \). The number of
these automata is $|aa(\forall x. \text{instrsOf}(P))| = |\text{instrsOf}(P)|^k$ for an axiom with $k$ universally quantified variables. Since this increases with $P$, the model is not finite state. Even so, this enables us to construct operational models for a given bound on $|\text{instrsOf}(P)|$. We can do this even for non-universal axioms by converting existential quantifiers into finite disjunctions over $\text{instrsOf}(P)$. We demonstrate an application of this in §VIII, where we check that a processor satisfies an axiom ensuring correctness of read values.

### C. Bounding the number of active instructions

As the discussion from §VII-B2 concludes, generating all concrete automata (statically) for arbitrary $\mu$spec specifications does not give us a finite state model. We need to bound the number of automata maintained at any point in the trace. In order to do this, for each index in the trace, we identify **active** instructions: an active instruction is one for which we need to maintain ordering information at that index. We observe that under the $t$-bounded reordering under-approximation, only a bounded number of instructions are active. This, in turn implies that we only need to maintain a bounded number of axiom automata. We now formalize these concepts.

For a $t$-reordering bounded trace $\sigma$ of a program $P$ and a trace index $0 \leq j \leq |\sigma|$, let $\text{CM}(j)$ and $\text{NF}(j)$ be instructions which have executed all and none of their events at $\sigma[j]$ respectively. We define the following auxiliary terms:

\[
\begin{align*}
\text{pCM}(j) & = \{ i \mid \forall i', i' \leq j \rightarrow i \in \text{CM}(j) \} \\
\text{pNF}(j) & = \{ i \mid \forall i', i \leq j \rightarrow i' \in \text{NF}(j) \} \\
\text{IP}(j) & = \text{instrsOf}(P) \setminus (\text{pCM}(j) \cup \text{pNF}(j))
\end{align*}
\]

Intuitively $\text{pCM}(j)$ represents the prefix-closed set of **completed** instructions, $\text{pNF}(j)$ represents the postfix-closed set of **not-fetched** instructions, and $\text{IP}(j)$ are the rest - the in-progress instructions (see Fig. 8). By the first condition of $t$-reordering boundedness, in-progress (IP) instructions on each core are bounded by $t$ for all $j$ (Lemma 4):

**Lemma 4.** For any $t$-reordering bounded trace $\sigma$, for all $0 \leq j \leq |\sigma|$, we have $|\text{IP}(j)| \leq |\text{Cores}| \cdot t$.

**Active instructions** Two instructions $i, i'$ are $k$-**coupled** in a trace $\sigma$ if they form a coupling chain of length $k$: i.e. there exist instructions $i_1, \ldots, i_{k-1}$ such that $\text{coup}(i_1, i_2), \ldots, \text{coup}(i_{k-1}, i')$. For trace $\sigma$, $0 \leq j \leq |\sigma|$ and $k \in \mathbb{N}$, we define $k$-**active** instructions at $j$, $\text{AC}_k(j)$, as instructions from $\text{pCM}(j) \cup \text{IP}(j)$ which are $k$-coupled with some instruction from $\text{IP}(j)$.

Intuitively, for a $\mu$specRE axiom with $k$ universally quantified variables, the execution of two instructions affect each other only if they are $k$-coupled. In particular, maintaining ordering information is important for instructions which are $k$-coupled with the in-progress instructions. As Lemma 5 shows, these **active** instructions - $\text{AC}_k(j)$ - are bounded at any given point in the trace.

**Lemma 5.** For each $k$, there is a (program-independent) bound $b_k$, s.t. for any $t$-reordering bounded trace $\sigma$, for all $0 \leq j \leq |\sigma|$, we have $|\text{AC}_k(j)| \leq b_k$.

The **operational model** Our operational model maintains the in-progress instructions (IP) on its tape. At each step it schedules an event from these instructions. The validity of event scheduling is ensured by maintaining orderings between events corresponding to the active instructions. Lemmas 4, 5 imply that at all points in the trace, (1) the set IP is bounded and (2) the active instructions - $\text{AC}_k$ - are bounded (as a function of $b_k$). Consequently, this results in a model which has finite state (used to maintain orderings between events of $\text{AC}_k$) and bounded history (owing to (1)). This gives us the main result - a finite state, bounded history operational model.

**Theorem 2.** For a (refinable) universal axiomatic semantics that satisfies extensibility, synthesis of finite-state, bounded-history operational models satisfying Def. 3 and 5* is feasible.

### VIII. Case Studies

In this section, we demonstrate applications of operationalization. We discuss three case studies: (1) multi_vscale [30] is a multi-core extension of the 3-stage in-order vscale [31] processor, (2) tomasulo is an OoO processor based on [32], and (3) sdram_ctrl is an SDRAM-controller [33].

For each case, we instrument the hardware designs by exposing ports that signal the execution of events (e.g. the PC ports in Fig. 9). We convert axioms into an operational model $\mathcal{M}$ based on the approach discussed in §VII. $\mathcal{M}$ is compiled to RTL and is synchronously composed with the hardware design, where it transitions on the exposed event signals. Thus, any violating behaviour of the hardware will lead $\mathcal{M}$ into a non-accepting (bad) state. Hence by specifying !bad as a safety property, we can perform verification of the RTL design w.r.t. the axioms. The operationalization approach enables us to perform both bounded and unbounded verification using off-the-shelf hardware model checkers. We highlight that this would not have been possible without operationalization.

We use the Yosys-based [34] SymbYosys as the model-checker, with booleogor [35] and abc [36] as backend solvers for BMC and PDR proof strategies respectively. Experiments
are performed on an Intel Core i7 machine with 16GB of RAM. We use our algorithm to automatically generate axiom automata. The compilation of the generated automata to RTL and their instrumentation with the design is done manually. However, in the future this could be automated following the procedure developed in §VII. The experimental designs are available at https://github.com/adwait/axiomatic-operational-examples.

Highlights. We demonstrate how the operationalization framework enables us to leverage off-the-shelf model checking tools implementing bounded and (especially) unbounded proof techniques such as IC3/PDR. This would not have been possible directly with axiomatic models. Even when Thm. 2 does not apply (e.g. non-universal/non-extensible axioms), following §VII-B2 we can fall back on a BMC-based check over all possible programs under a bound on |instrsOf(P)|.

A. The multi_vscale processor

a) Pipeline axioms on a single core: We begin with the single-core variant of multi_vscale. We are interested in verifying the pipeline axioms for this core. The first axiom states that pipeline stages must be in Fet-DX-WB order and the second enforces in-order fetch.

\[ \forall i1. (hb(i1.Fet, i1.DX) \land hb(i1.DX, i1.WB)) \]

\[ \forall i1, i2. i1 < i2 \Rightarrow hb(i1.Fet, i2.Fet) \]

The setup schematic is given in Figure 9. \( \mathcal{M} \) is the operational model implemented in RTL (note that we could do this only because the model is finite state and requires a finite history \( h \)). Given that it is a 3-stage in-order processor, at any given point each core has at most 3 instructions in its pipeline and we can safely choose a history parameter of \( h = 3 \), and \( \mathcal{M} \) is complete for a reordering bound of \( t = 3 \). We replace the \texttt{imem_hrdata} (instruction data) connection to the core by an input signal that we can symbolically constrain. Using this input signal, we can control the program (instruction stream) executed by the core.

Verification is performed with a PDR based proof using the abc pdr backend. We experiment with various choices of instructions fed to the processor (by symbolically constraining \texttt{imem_hrdata}). In Fig. 10, we show the constraint and its PDR proof runtime, with BMC runtime (depth = 20) for comparison. These examples demonstrate our ability to prove unbounded correctness.

b) Memory ordering on multi-core: We now configure the design with 2 cores: \( c_0, c_1 \), both initialized with symbolic load and store operations. We then perform verification w.r.t. the \texttt{ReadValues} (RV) axiom shown below. This axiom says that for any read instruction \( i1 \), the value read should be the same as the most recent write instruction \( i2 \) on the same address, or it should be the initial value.

\[ \forall i1, \exists i2, \forall i3. \text{IsRead}(i1) \implies (\text{DataInit}(i1) \lor (\text{IsWrite}(i2) \land \text{SameAddr}(i1, i2) \land \text{ValEq}(i1, i2) \land ((\text{IsWrite}(i3) \land \text{SameAddr}(i1, i3)) \implies (\text{ValEq}(i1, i3) \lor \text{hb}(i3.DX, i2.DX) \lor \text{hb}(i1.DX, i3.DX))))\]

This not a universal axiom, and hence Thm. 2 does not apply. However, for bounded programs we can construct \(|\text{instrsOf}(P)|^2\) concrete automata (since there are two universally quantified variables: \( i1, i3 \)) as discussed in §VII-B2. We convert the existential quantifier over \( i2 \) into a finite disjunction over \( \text{instrsOf}(P) \). We perform BMC queries for programs with \(|I| = |\text{instrsOf}(P)| = 4, 6, 8\).

By keeping instructions symbolic, we effectively prove correctness for all programs within our bound \(|I|\). The table alongside shows the instruction bound, \(|I|\), the number of axiom automata \(|\mathcal{A}|\), BMC depth \( d \), and proof runtime. Though our theoretical results apply to universal axioms, this shows how an axiom automata-based operationalization can be applied to arbitrary axioms by bounding \(|\text{instrsOf}(P)|\).

B. An OoO processor: tomasulo

Our second design is an out-of-order processor (based on [32]) that implements Tomasulo’s algorithm. The processor has stages: \( F \) (fetch), \( D \) (dispatch), \( I \) (issue), \( E \) (execute), \( WB \) (writeback), and \( C \) (commit). We verify in-order-commit, program-order fetch, and pipeline order axioms for this processor. A BMC proof (with \( d = 20 \)) takes ~2m.

The axiom \texttt{axDep} given below is crucial for correct execution in an OoO processor. It enforces that execute (E) stages for consecutive instructions should be in program order if the destination of the first instruction is same as the source of the second, i.e. dependent instructions are executed in order.

\[ \forall i1, i2, (i1 < i2 \land \text{Cons}(i1, i2) \land \text{DepOn}(i1, i2)) \implies \text{hb}(i1.E, i2.E) \]

We add a program counter (pc) to instructions and define \texttt{Cons(i1,i2)} \( \equiv \text{pc}(i1) + 4 = \text{pc}(i2) \) and \texttt{DepOn(i1,i2)} \( \equiv \text{dest}(i1) = \text{src1}(i1) \lor \text{dest}(i1) = \text{src2}(i2) \).

As before, we compose the operational model \( \mathcal{M} \) corresponding to this axiom with the RTL design. We symbolically constrain the processor to execute a sequence of symbolic (add and sub) instructions and assert !bad. A BMC query
results in an assertion violation. We manually identified the bug as being caused by the incorrect reset of entries in the Register Alias Table (RAT) in the Com stage. When committing instruction \(i_0\), the entry RAT\(\text{dest}(i_0)\) is reset, while some instruction \(i_1\) with \(\text{dest}(i_1) = \text{dest}(i_0)\) is issued at the same cycle. A third instruction \(i_2\) with \(\text{src}(i_2) = \text{dest}(i_0)\) then reads the result of \(i_0\) instead of \(i_1\), violating the axiom. We fix this bug and perform a BMC proof \((d = 20)\), which takes \(\sim 6m30s\). This demonstrates how our technique can be used to identify a bug, correct it and check the fixed design.

C. A memory controller: sdram_ctrl

To demonstrate the versatility of our approach, we experiment with an SDRAM controller [33], which interfaces a processor host with an SDRAM device, with a ready-valid interface for read/write requests. All intricacies related to interfacing with the SDRAM are handled by maintaining appropriate control state in the controller. In the following, we once again convert axioms into an operational model by our technique, and compose the generated model with the design.

First we verify pipeline-stage axioms for sdram_ctrl for write (4-stages) and read (5-stages) operations executed by the host. A PDR-based (unbounded) proof for the pipeline axioms requires \(\sim 8m\). Next we verify properties related to SDRAMs refresh operation [37]. The controller ensures that the host-level behaviour is not affected by refreshes by creating an illusion of atomicity for writes and reads. This results in the axiom that once a write or read operation is underway, no refresh stage should execute before it is completed. We once again prove this property with PDR, which takes \(\sim 1m30s\).

IX. Related work

There has been much work on developing axiomatic (declarative) models for memory consistency in parallel systems, at the ISA level [2], [38], [39], the microarchitectural level [12], [16], [11], and the programming language level [20], [40], [41], [42], [43]. There has also been work on constructing equivalent operationalizations for these models, e.g., for Power [2], ARMv8 [10], RA[8], C++ [7], and TSO [19], [9]. These constructions are accompanied by hand-written/theorem-prover based proofs, demonstrating equivalence with the axiomatic model. In principle, our work is related to these, however we enable automatic generation of equivalent operational models from axiomatic ones, eliminating most of the manual effort.

At an abstract level, we have been inspired by classic works that have developed connections between logics and automata [44], [45]. There is a large body of work on synthesis of operational implementations as well as monitors from temporal specifications (e.g. [46], [47], [48]), most commonly those written in Linear Temporal Logic (LTL) [49] and its variants (e.g. [50]). In this paper we perform a similar conversion but for a very different logic: \(\mu\text{spec}\) constrains orderings of a known set of events, while LTL does so over traces with potentially differing sets of events (atoms). These differences make a direct comparison with the previously mentioned works ineffectual, and have required us to develop novel concepts in this work.

In terms of the application to proving properties, the work closest to ours is RTLCheck [13], which compiles constraints from \(\mu\text{spec}\) to SystemVerilog assertions. These assertions are checked on a per-program basis. On the other hand, we demonstrate the ability to prove unbounded correctness. Additionally, for axioms that are not generally operationalizable (for unbounded programs), we demonstrate the ability to generate an operational model for some apriori known bound on the program size. In this case, we can verify correctness for all programs of size up to that bound, as opposed to on a per-program basis as RTLCheck does. RTL2\(\mu\text{spec}\) [51] aims to perform the reverse conversion: from RTL to \(\mu\text{spec}\) axioms.

X. Conclusion

In this paper we make strides towards enabling greater interoperability between operational and axiomatic models, both through theoretical results and case studies. We derive \(\mu\text{specRE},\) a restricted subset of the \(\mu\text{spec}\) domain-specific language for axiomatic modelling. We show that the generation of an equivalent finite-state operational model is impossible for general \(\mu\text{spec}\) axioms, though it is feasible for universal axioms in \(\mu\text{specRE}\). From a practical standpoint, we develop an approach based on axiom automata that enables us to automatically generate such equivalent operational models for universally quantified axioms in \(\mu\text{specRE}\) (or for arbitrary \(\mu\text{spec}\) axioms if equivalence up to a bound is sufficient).

The challenges we surmount for our conversion (discussed in §I) find parallels in manual operationalization works [7], and we believe that the above concepts can be extended to formalisms such as Cat [2]. Our practical evaluation illustrates the key impact of this work—its ability to enable users of axiomatic models to take advantage of the vast number of techniques that have been developed for operational models in the fields of formal verification and synthesis.

XI. Future Work

An interesting direction for future work is to enrich \(\mu\text{spec}\) semantics (e.g., with quantitative operators) such that valid executions are guaranteed to satisfy \(\mu\text{spec}\) reordering boundedness. In addition to allowing generation of finite-state operational models, we believe that such axioms would also capture processor executions more precisely.

While some aspects of executions are easier to specify operationally, others (e.g., non-deterministic scheduling) are better suited to axiomatic specifications. Another direction for future work is combining operational and axiomatic modelling, for example using tools such as UCLID5 [52], [53].

Acknowledgments

This work was supported in part by Intel under the Scalable Assurance program and by DARPA contract FA8750-20-C-0156.