Formal Methods in Computer-Aided Design 2022

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 models 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_ctrl`), and an out-of-order single-core processor (`tomasulo`). 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.

https://doi.org/

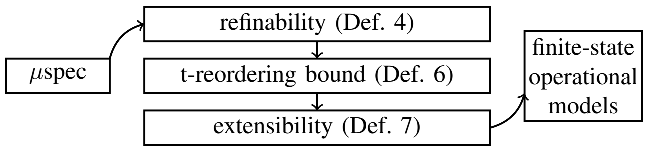


Fig. 1: Roadmap to obtain finite-state operational models.

- 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_vscale`, `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

$$
\begin{aligned}
\langle \mathtt{AX} \rangle &:= \forall \mathtt{i}\ \ \mathtt{AX} \mid \exists \mathtt{i}\ \ \mathtt{AX} \mid \phi(\mathtt{i}_1, \cdots, \mathtt{i}_m) \\
\langle \phi \rangle &:= \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \langle \mathsf{atom} \rangle \\
\langle \mathsf{atom} \rangle &:= \mathtt{i}_1 <_r \mathtt{i}_2 \mid \mathsf{hb}(\mathtt{i}_1.\mathsf{st}, \mathtt{i}_2.\mathsf{st}) \mid \mathsf{P}(\mathtt{i}_1, \dots) \\
\langle \mathsf{st} \rangle &:= \mathtt{Fet} \mid \mathtt{Dec} \mid \mathtt{Exe} \mid \mathtt{WB} \mid \cdots
\end{aligned}
$$

Fig. 2: μspec Syntax.

μspec [11] is a domain-specific language used for specifying microarchitectural orderings. A μspec model consists of *axioms* that enforce first-order constraints over execution graphs; each axiom quantifies over instructions and is required to be a sentence (i.e. not have any free variables). Execution graphs that satisfy the axioms and are acyclic are deemed as valid executions. While ISA-level models [2], [18], [19] treat single instructions as atomic entities, μspec decomposes the execution of an instruction into a set of atomic *events*. Each instruction i and stage st is associated with an event i.st. A program execution is viewed as a directed acyclic graph called a micro-architectural happens-before graph (μhb graph) [12]. Such a graph for a given program has nodes corresponding to events of form i.st for each instruction i in the program and each stage st prescribed by the model. Edges in the graph correspond to the *happens-before* (hb) relation: $\mathsf{hb}(\mathsf{e}_1, \mathsf{e}_2)$ says that e_1 happened before e_2. Thus, a cyclic μhb graph corresponds to an impossible scenario where an event happens before itself, and thus represents an execution that cannot occur on the microarchitecture.

Fig. 2 specifies μspec syntax. It has three types of atoms:

(i) $\mathsf{hb}(\mathtt{i}_1.\mathsf{st}, \mathtt{i}_2.\mathsf{st})$: happens-before predicate
(ii) $\mathsf{i}_1 <_r \mathsf{i}_2$: the reference order (typically the program order)
(iii) $\mathsf{P}(\mathtt{i}_1, \dots)$: instruction predicate atoms

Atoms of type (ii) capture the order in which instructions appear in a given program thread. Atoms of type (iii) are predicates over instructions which capture instruction properties, e.g. opcode, source/destination registers. We note that μspec models in literature [11] also make use of the `NodeExists` predicate which identifies event nodes that occur in the execution. We do not model `NodeExists` in this paper, but our approach can be augmented to incorporate it (see [17]).

We identify two types of axioms of interest: *Universal axioms* are of the form: $\forall \mathtt{i}_1 \cdots \forall \mathtt{i}_k\ \phi(\mathtt{i}_1, \cdots, \mathtt{i}_k)$, and represent constraints applied symmetrically over all tuples of instructions in a program. *Predicate-free axioms* are axioms that do not have occurrences of predicate (P) atoms. We extend these terms to an axiomatic semantics if all axioms are of that type. In this work, our theoretical treatment focuses on universal semantics. Practically though, some underlying ideas carry over to arbitrary axioms as we discuss in §VII, §VIII.

```
ax0: ∀ i1. hb(i1.Exe,i1.Com)
ax1: ∀ i1,i2. (i1<_r i2 ∧ DepOn(i1,i2))
      ⟹ hb(i1.Exe,i2.Exe)
ax2: ∀ i1,i2. SameCore(i1,i2) ⇒
(hb(i1.Exe,i2.Exe)∨hb(i2.Exe,i1.Exe))
ax3: ∀ i1,i2. i1<_r i2 ⇒ hb(i1.Com,i2.Com)
```

Fig. 3: An example axiomatic model.

B. Illustrative μspec Example

Consider the four axioms in Fig. 3. In the axioms, **i1**, **i2** are instruction variables and **Exe**,**Com** are stage names (short for execute and commit respectively). The axiom **ax0** requires that for each instruction, the execute stage (**Exe**) of that instruction must happen before the commit stage (**Com**). Intuitively, **ax1** says that when **i2** depends on **i1** (captured by the predicate **DepOn**) , **i1** should be executed before **i2**; **ax2** says that the execute events of instructions on the same core should be *totally ordered* by hb. The third axiom **ax3** says that when **i1** and **i2** are in program order (denoted by $<_r$), **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 i_1 is dependent on the result of i_0 (since its source register is the same as the destination of i_0). In the example axiomatic semantics, **ax1** requires that the execute event of instruction i_0 be before that of i_1. The execution in Fig. 4b is invalid w.r.t. **ax1** since i_1.**Exe** is executed before i_0.**Exe**. The execution in Fig. 4a is valid even though the i_2.**Exe** and i_3.**Exe** events are reordered since i_3 does not depend on i_2. 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: $\mathsf{Cores} = [n]$. Each core executes operations from a *finite* set $\mathbb{O}$. The axiomatic model $\mathcal{A}$ assigns predicates from $\mathbb{P}$ an interpretation over the universe $\mathbb{O}$. We denote this interpretation as $\mathsf{P}^{\mathcal{A}} \subseteq \mathbb{O}^k$ for an arity-k predicate.

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

[2]Note the terminology: operations are commands that the core can execute. Since we interpret predicates over $\mathbb{O}$ we require $|\mathbb{O}|$ to be a finite set for computability reasons. Instructions are operations combined with the label and core identifier (and hence form an infinite set).

3) Instruction stages: Instruction execution in μspec is decomposed into stages. The set of stages, 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 $\mathcal{P}$ is composed of the set of events: $\mathsf{eventsOf}(\mathcal{P}) = \{\mathsf{i.st} \mid \mathsf{i} \in \mathsf{instrsOf}(\mathcal{P}), \mathsf{st} \in \mathsf{Stages}\}$. The set of all possible events is $\mathbb{E} = \{\mathsf{i.st} \mid \mathsf{i} \in \mathbb{I}, \mathsf{st} \in \mathsf{Stages}\}$.

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

Example 1. *Following the example in Fig. 5 we consider an architecture with two opcodes:* **add**, **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* $\mathcal{P}$ *in Fig. 5 has two cores:* $\mathsf{Cores} = \{\mathsf{c}_0, \mathsf{c}_1\}$ *and four instructions:* $\mathsf{instrsOf}(\mathcal{P}) = \{\mathsf{i}_0, \mathsf{i}_1, \mathsf{i}_2, \mathsf{i}_3\}$*. We have, for example,* $c(\mathsf{i}_1) = \mathsf{c}_0, \lambda(\mathsf{i}_1) = 1, op(\mathsf{i}_1) =$ **add r3, r2, r1** *while* $c(\mathsf{i}_2) = \mathsf{c}_1, \lambda(\mathsf{i}_2) = 0$*. The instruction stream for core* c_0 *is* $\mathcal{I}_0 = \mathsf{i}_0 \cdot \mathsf{i}_1$ *anf that of core* c_1 *is* $\mathcal{I}_1 = \mathsf{i}_2 \cdot \mathsf{i}_3$*.*

Let us suppose that this program is executed on a 4-stage microarchitecture with $\mathsf{Stages} = \{$**Fet**, **Dec**, **Exe**, **WB**, **Com**$\}$*. The events corresponding to the program are given by* $\mathsf{eventsOf}(\mathcal{P}) = \{\mathsf{i}_0.$**Fet**$, \mathsf{i}_0.$**Dec**$, \cdots, \mathsf{i}_3.$***Com***$\}$ *with* $|\mathsf{eventsOf}(\mathcal{P})| = 4 \times 5 = 20$.

D. Formal μspec Semantics

We now define the formal semantics of μspec axioms.

Definition 2 (μhb graph)**.** *For a program* $\mathcal{P}$*, a* μhb *graph is a directed acyclic graph,* $G(V, E)$*, with nodes* $V = \mathsf{eventsOf}(\mathcal{P})$ *representing events and edges representing the happens-before relationships, i.e.* $(\mathsf{e}_1, \mathsf{e}_2) \in E \equiv \mathsf{hb}(\mathsf{e}_1, \mathsf{e}_2)$.

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

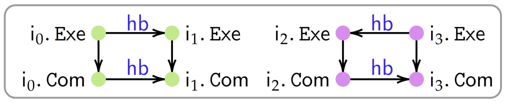
Satisfaction w.r.t. an axiom: We first define satisfaction for the quantifier-free part, starting at the atoms. Let $s : \mathtt{I}(\mathrm{AX}) \to \mathbb{I}$ be an assignment for the symbolic instruction variables $\mathtt{I}(\mathrm{AX})$ in axiom AX.

$$G \models \mathtt{i}_1[s] <_r \mathtt{i}_2[s] \iff \mathsf{c}(s(\mathtt{i}_1)) = \mathsf{c}(s(\mathtt{i}_2)) \wedge \lambda(s(\mathtt{i}_1)) < \lambda(s(\mathtt{i}_2)) \quad \ldots(i)$$

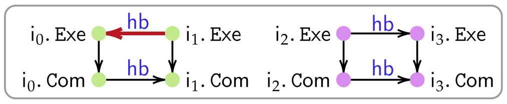
$$G \models \mathsf{P}(\mathtt{i}_1, \cdots, \mathtt{i}_m)[s] \iff (\mathsf{op}(s(\mathtt{i}_1)), \cdots, \mathsf{op}(s(\mathtt{i}_m))) \in \mathsf{P}^{\mathcal{A}} \quad \ldots(ii)$$

$$G \models \mathsf{hb}(\mathtt{i}_1.\mathsf{st}_1, \mathtt{i}_2.\mathsf{st}_2)[s] \iff (s(\mathtt{i}_1).\mathsf{st}_1, s(\mathtt{i}_2).\mathsf{st}_2) \in E^+ \quad \ldots(iii)$$

In *(i)*, the reference order $<_r$ relates instructions $\mathtt{i}_1, \mathtt{i}_2$ from the same instruction stream if $\mathtt{i}_1$ is before $\mathtt{i}_2$. In *(ii)* we extend predicate interpretations, $\mathsf{P}^{\mathcal{A}}$, (defined over $\mathbb{O}$) to instructions by taking the $\mathsf{op}(\cdot)$ component. Finally, hb atoms



(a)



(b)

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 hb relation. The red (bold) edge violates **ax1**.

```
i0: lw r1, 42(r0)     ||  i2: lw r4, 42(r0)
i1: add r3, r2, r1    ||  i3: add r3, r2, r1
```

Fig. 5: Example program snippet

are interpreted as E^+, i.e. transitive closure of E, as stated in *(iii)*. Operators $\wedge, \vee, \neg$ have their usual semantics.

We now define the satisfaction of a (quantified) axiom AX by a graph G, denoted by $G \models_{\mathcal{P}}$ AX above.

$$\begin{aligned} G \models_{\mathcal{P}} \phi[s] \quad &\equiv G \models \phi[s] \\ &\quad \text{for quantifier-free } \phi \\ G \models_{\mathcal{P}} \forall \mathtt{i}\ \phi[s] \quad &\equiv G \models_{\mathcal{P}} \phi[s[\mathtt{i} \leftarrow \mathsf{i}]] \\ &\quad \text{for all } \mathsf{i} \in \mathsf{instrsOf}(\mathcal{P}) \setminus \mathit{range}(s) \\ G \models_{\mathcal{P}} \exists \mathtt{i}\ \phi[s] \quad &\equiv G \models_{\mathcal{P}} \phi[s[\mathtt{i} \leftarrow \mathsf{i}]] \\ &\quad \text{for some } \mathsf{i} \in \mathsf{instrsOf}(\mathcal{P}) \setminus \mathit{range}(s) \end{aligned}$$

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

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 under-approximate, 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 Cores, stages 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 $(\mathcal{Q}, \Delta, \mathsf{q}_{\mathsf{init}}, \mathsf{q}_{\mathsf{final}})$:

- $\mathcal{Q}$ is a finite set of control states
- $\Delta \subseteq \mathcal{Q} \times (\mathbb{I} \cup \{\dashv\})^{|\mathsf{Cores}|} \times \mathcal{Q} \times \mathsf{Act}$ is the transition relation where Act is the set of actions
- $\mathsf{q}_{\mathsf{init}} \in \mathcal{Q}$ is the initial state
- $\mathsf{q}_{\mathsf{final}} \in \mathcal{Q}$ is the final state which must be absorbing (i.e. it has a self-loop)

[3]A Kripke structure-based formalism is insufficient since we want to execute unbounded programs with distinguished instructions without explicitly modelling control logic.

A model is finite-state if $\mathcal{Q}$ 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 $\gamma = (\mathsf{U}, \mathsf{q}, \mathsf{V})$ where $\mathsf{U} : \mathsf{Cores} \to \mathbb{I}^*$, $\mathsf{V} : \mathsf{Cores} \to \mathbb{I}^*$ and $\mathsf{q} \in \mathcal{Q}$. 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 $|\mathsf{U}(\mathsf{c})| \leq h$ for all $\mathsf{c} \in \mathsf{Cores}$. For unbounded history all configurations are allowed.

The set of actions is

$$\begin{aligned} \mathsf{Act} = \ & \{\mathsf{right}(\mathsf{c}) \mid \mathsf{c} \in \mathsf{Cores}\} \cup \\ & \{\mathsf{stay}\} \cup \\ & \{\mathsf{sched}(\mathsf{c}, i, \mathsf{st}) \mid \mathsf{c} \in \mathsf{Cores}, \mathsf{st} \in \mathsf{Stages}, i \in [h]\} \cup \\ & \{\mathsf{drop}(\mathsf{c}, i) \mid \mathsf{c} \in \mathsf{Cores}, i \in [h]\} \end{aligned}$$

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 $\mathtt{fst}(w)$ denote its first element if $w \neq \epsilon$ and $\dashv$ otherwise. Transitions are enabled based on the control state and the instructions that the tape-heads point to: transition $(\mathsf{q}_1, (\mathsf{i}_1, \cdots, \mathsf{i}_{|\mathsf{Cores}|}), \mathsf{q}_2, \_) \in \Delta$ is enabled in configuration $\gamma = (\mathsf{U}, \mathsf{q}, \mathsf{V})$ if $\mathsf{q}_1 = \mathsf{q}$ and $\mathtt{fst}(\mathsf{V}(\mathsf{c})) = \mathsf{i}_\mathsf{c}$ for each $\mathsf{c} \in \mathsf{Cores}$.

3) Runs: The initial configuration is given by $\gamma_{\mathsf{init}}(\mathcal{P}) = (\mathsf{U}_{\mathsf{init}}, \mathsf{q}_{\mathsf{init}}, \mathsf{V}_{\mathsf{init}})$ where $\mathsf{U}_{\mathsf{init}} = \lambda\mathsf{c}.\ \epsilon$ and $\mathsf{V}_{\mathsf{init}} = \lambda\mathsf{c}.\ \mathcal{I}_\mathsf{c}$, 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 $\gamma_{\mathsf{init}}(\mathcal{P})$, the machine transitions according to the transition rules. Such a sequence of configurations $\gamma_{\mathsf{init}}(\mathcal{P}) = \gamma_0 \xrightarrow{\mathsf{e}_1} \gamma_1 \cdots \xrightarrow{\mathsf{e}_m} \gamma_m$, where all γ_i are allowed is called a run. A run is called accepting if it ends in the state $\mathsf{q}_{\mathsf{final}}$.

4) Traces: The sequence of event labels $\sigma = \mathsf{e}_1 \cdots \mathsf{e}_m$ annotating a run is the *trace* corresponding to the run. Each label is an event from $\mathbb{E}$ and hence $\sigma \in \mathbb{E}^*$. We view σ as a (linear) μhb execution graph $\mathsf{e}_1 \xrightarrow{\mathsf{hb}} \mathsf{e}_2 \cdots \xrightarrow{\mathsf{hb}} \mathsf{e}_m$, and hence define $\sigma \models \mathcal{A}$ in the usual way. Accordingly, we will sometimes refer to σ as an execution of a program $\mathcal{P}$. The set of traces corresponding to accepting runs of an operational model $\mathcal{M}$ on a program $\mathcal{P}$ are denoted as $\mathsf{traces}_{\mathcal{M}}(\mathcal{P}) \subseteq \mathbb{E}^*$.

IV. Soundness, Completeness, and Equivalence

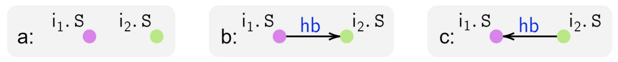
We proceed to formalize the notion of equivalence that relates axiomatic and operational models. In literature [29], [2], ISA-level behaviours of programs have been annotated by the read values of `load` operations. Hence, one notion of equivalence might be to require that identical read values be possible 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 μspec 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 μhb graphs that are valid under the axiomatic semantics. Formally:

Definition 3 (Soundness)**.** *An operational model $\mathcal{M}$ is sound w.r.t. $\mathcal{A}$ if for any program $\mathcal{P}$, each trace in $\mathsf{traces}_{\mathcal{M}}(\mathcal{P})$ is a linearization of some μhb graph that is valid under $\mathcal{A}$.*

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

Example 2 (Non-refinable axiom)**.** For the following axiom with $\mathsf{Stages} = \{\mathtt{S}\}$, the graph (a) is a valid execution. However, both of its linearizations (b) and (c) are invalid. Thus, *all* of the (totally-ordered) traces generated by our operational models will be deemed invalid under the axiomatic semantics. This renders a direct comparison between operational and axiomatic executions infeasible.

$$\forall\ \mathtt{i1,i2.}(\neg\mathsf{hb}(\mathtt{i1.S,i2.S})\wedge\neg\mathsf{hb}(\mathtt{i2.S,i1.S}))$$



To address this issue, we develop the notion of refinability. For two μhb graphs $G = (V, E)$ and $G' = (V', E')$, we say that G' refines G, denoted $G \sqsubseteq G'$ if (1) $V = V'$ and (2) $(\mathsf{e}_1, \mathsf{e}_2) \in E^+ \implies (\mathsf{e}_1, \mathsf{e}_2) \in E'^+$.

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

Refinability says that all linearizations of a valid graph are valid. While executions under axiomatic semantics are given by (partially-ordered) μhb graphs, our class of operational models generate totally-ordered traces. Refinability bridges this gap by relating valid μhb graphs to valid traces. Interestingly, we can check whether a universal axiomatic semantics satifies 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 $\mathcal{M}$ is complete, if for any program $\mathcal{P}$ and valid μhb graph $G \models_{\mathcal{P}} \mathcal{A}$, $\mathsf{traces}_{\mathcal{M}}(\mathcal{P})$ contains all linearizations of G.*

Formal Problem Statement Given an axiomatic semantics $\mathcal{A}$, a set of cores Cores and stages Stages, generate a *finite state, bounded history* model, $\mathcal{M} = (\mathcal{Q}, \Delta, \mathsf{q}_{\mathsf{init}}, \mathsf{q}_{\mathsf{final}})$, which satisfies soundness and completeness (Defns. 3 and 5).

V. Enabling Synthesis by Bounding Reorderings

In this section, we develop some theoretical results for the synthesis of operational models. First, we show that synthesis of sound and complete (viz. Defn. 3 and 5) finite-state operational models is not possible. Then we provide an underapproximation for the completeness requirement, called t-completeness, that enables the synthesis of finite-state models. This still does not allow for bounded-history models as future events can influence past orderings (Example 3). In §VI we add *extensibility* thus enabling our original goal of finite-state and bounded-history models.

A. An impossibility result

We show that it is in fact impossible to develop a finite-state transition system $\mathcal{M}$ that satisfies the requirements prescribed in Defns. 3 and 5. Figure 6 gives an axiomatic semantics $\mathcal{A}^{\#}$ (with $\mathsf{Stages} = \{\mathtt{S}, \mathtt{T}\}$) such that for all possible finite-state models, there is some program such that either soundness or completeness is violated. In words, the axioms in Fig. 6 state

```
ax0: ∀ i1.     hb(i1.S,i1.T)
ax1: ∀ i1,i2. hb(i1.S,i2.S)⇒hb(i1.T,i2.T)
```

Fig. 6: Semantics $\mathcal{A}^{\#}$ that does not allow bounded synthesis

the following constraints: **ax0** says that for each instruction, the **S** stage event happens before the **T** stage, and **ax1** enforces that for any two instructions, the ordering between their **S** stage events implies an identical ordering between their **T** stage events. We have the following:

Theorem 1. *For a single-core program $\mathcal{P}$ with an instruction stream of $|\mathcal{I}_{\mathsf{c}_1}| = m$ instructions, there is no model $\mathcal{M} = (\mathcal{Q}, \Delta, \mathsf{q}_{\mathsf{init}}, \mathsf{q}_{\mathsf{final}})$ that is sound and complete w.r.t. $\mathcal{A}^{\#}$ and $\mathcal{P}$, and s.t. $|\mathcal{Q}| < \mathcal{O}(2^m/m)$, even with $h = \infty$.*

We provide an intuitive explanation, deferring details to the supplement [17]. In valid executions of $\mathcal{A}^{\#}$, **S** stage events 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 $\mathcal{A}^{\#}$ 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 $\mathsf{i}_1, \mathsf{i}_2$ on the same core, let $\mathsf{diff}_r(\mathsf{i}_1, \mathsf{i}_2) = \lambda(\mathsf{i}_2) - \lambda(\mathsf{i}_1)$ (recall that $\lambda(\mathsf{i})$ is the instruction index of i). Consider a trace σ of program $\mathcal{P}$. For $\mathsf{i} \in \mathsf{instrsOf}(\mathcal{P})$, we define the starting index of i, denoted as $\mathsf{start}(\mathsf{i})$, as the index of the first event of instruction i in σ. Similarly we define the ending index, $\mathsf{end}(\mathsf{i})$ as the largest index for some event of i in σ. Let the *prefix-closed end index* of i be the max of end over instructions that are $\leq_r$ i: $\mathsf{pfxend}(\mathsf{i}) = \max\{\mathsf{end}(\mathsf{i}') \mid \mathsf{i}' \leq_r \mathsf{i}\}$. Two instructions i_1 and i_2 are coupled in a trace (denoted as $\mathsf{coup}(\mathsf{i}_1, \mathsf{i}_2)$) if the intervals $[\mathsf{start}(\mathsf{i}_1), \mathsf{pfxend}(\mathsf{i}_1)], [\mathsf{start}(\mathsf{i}_2), \mathsf{pfxend}(\mathsf{i}_2)]$ overlap.

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

Intuitively, (1) says that an instruction cannot be reordered with another that precedes it by $\geq t$ indices, while (2) says that instructions on a core cannot be *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 $\mathcal{M}$ is t-complete w.r.t. an axiomatic model $\mathcal{A}$, if for each program $\mathcal{P}$ and $G \models_{\mathcal{P}} \mathcal{A}$, $\mathsf{traces}_{\mathcal{M}}(\mathcal{P})$ 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 implementations in practice, in addition to finite-state, we also require bounded-history ($h \in \mathbb{N}$). This is addressed in the next section.

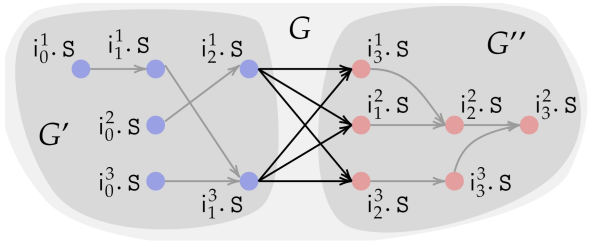

G
G'
G''
$\mathsf{i}^1_0.\mathsf{S}$
$\mathsf{i}^1_1.\mathsf{S}$
$\mathsf{i}^1_2.\mathsf{S}$
$\mathsf{i}^2_0.\mathsf{S}$
$\mathsf{i}^3_0.\mathsf{S}$
$\mathsf{i}^3_1.\mathsf{S}$
$\mathsf{i}^1_3.\mathsf{S}$
$\mathsf{i}^2_1.\mathsf{S}$
$\mathsf{i}^2_2.\mathsf{S}$
$\mathsf{i}^2_3.\mathsf{S}$
$\mathsf{i}^3_3.\mathsf{S}$
$\mathsf{i}^3_2.\mathsf{S}$

Fig. 7: $\mathcal{P}$ has instruction streams $\mathsf{i}^1_0 \cdot \mathsf{i}^1_1 \cdot \mathsf{i}^1_2 \cdot \mathsf{i}^1_3$, $\mathsf{i}^2_0 \cdot \mathsf{i}^2_1 \cdot \mathsf{i}^2_2 \cdot \mathsf{i}^2_3$, and $\mathsf{i}^3_0 \cdot \mathsf{i}^3_1 \cdot \mathsf{i}^3_2 \cdot \mathsf{i}^3_3$. Blue instructions form the prefix $\mathcal{P}'$ (i.e. $\mathcal{P}' \preceq \mathcal{P}$) and red its residual $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$. The figure shows executions G' of $\mathcal{P}'$, G'' of $\mathcal{P}''$, and their composition $G = G' \triangleright G''$.

VI. Adding Extensibility

As illustrated by the following example, the t-reordering bounded underapproximation is insufficient to achieve bounded-history operational model synthesis on its own.

Example 3 (Need for extensibility)**.** Consider a single stage axiomatic semantics: $\mathsf{Stages} = \{\mathbf{S}\}$, and predicate $\mathbb{P} = \{\mathsf{P}\}$.

$$\forall\ \mathtt{i0,i1,i2}.\ \ (\mathsf{P}(\mathtt{i0,i1,i2})\ \wedge\ \mathtt{i0}<_r\mathtt{i1}) \\ \implies\ \neg\mathsf{hb}(\mathtt{i1.S,i0.S})$$

There cannot be a sound, t-complete, and bounded-history (for bound h) model for this axiom (for some $t > 1$). To see this, consider a (single-core) program $\mathcal{P}$, with instructions $\mathsf{i}_0 \cdot \mathsf{i}_1 \cdots \mathsf{i}_{h+1}$. Depending on the instructions in $\mathcal{P}$, the interpretation $\mathsf{P}^{\mathcal{A}}$ of P can either be (a) $\mathsf{P}^{\mathcal{A}} = \{(\mathsf{i}_0, \mathsf{i}_1, \mathsf{i}_{h+1})\}$ or (b) $\mathsf{P}^{\mathcal{A}} = \{\}$. In the former case, the ordering $\mathsf{i}_1.\mathbf{S} \xrightarrow{\mathsf{hb}} \mathsf{i}_0.\mathbf{S}$ is invalid while in the latter it is valid. Since we only allow a h-sized history, $\mathsf{i}_0.\mathbf{S}$ must be scheduled before the tape-head reaches i_{h+1}, i.e. before the machine can determine which of (a)/(b) hold. Since the machine cannot determine whether events $\mathsf{i}_0.\mathbf{S}$, $\mathsf{i}_1.\mathbf{S}$ can be reordered, this leads either to a model which is unsound (always reorders) or incomplete (never reorders).

Thus, we need an additional restriction to enable generation of operational models with a finite history parameter h. We propose extensibility, which intuitively states that partial executions of program $\mathcal{P}$ that have not violated any axioms can be composed with valid executions of the residual program to generate valid complete executions of $\mathcal{P}$. To do this, we extend the notion of validity to partial executions through *prefix programs*.

A program $\mathcal{P}$ can be split into a prefix $\mathcal{P}'$ (blue) and the residual suffix $\mathcal{P}''$ (red) (Fig. 7). Formally, $\mathcal{P}'$ is a *prefix* of program $\mathcal{P}$, if $\mathcal{P}'$ has instruction streams $\{\mathcal{I}'_i\}$, each of which is a prefix of the instr. streams $\{\mathcal{I}_i\}$ of $\mathcal{P}$. We denote that $\mathcal{P}'$ is a prefix of $\mathcal{P}$ by $\mathcal{P}' \preceq \mathcal{P}$. For programs $\mathcal{P}, \mathcal{P}'$ such that $\mathcal{P}' \preceq \mathcal{P}$ we denote the *residual* of $\mathcal{P}$ w.r.t. $\mathcal{P}'$ as $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$. $\mathcal{P}'$ has instr. streams $\mathcal{I}''_{\mathsf{c}}$: for each core c, $\mathcal{I}_{\mathsf{c}} = \mathcal{I}'_{\mathsf{c}} \cdot \mathcal{I}''_{\mathsf{c}}$.

In Fig. 7, for example, the first instruction stream of $\mathcal{P}$ is $\mathsf{i}^1_0 \cdot \mathsf{i}^1_1 \cdot \mathsf{i}^1_2 \cdot \mathsf{i}^1_3$. The prefix program $\mathcal{P}'$ has (the prefix) $\mathsf{i}^1_0 \cdot \mathsf{i}^1_1 \cdot \mathsf{i}^1_2$ as its first instr. stream. On the other hand, the residual program, $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$, has the suffix i^1_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) $E = E' \cup E'' \cup \{(\mathsf{e}', \mathsf{e}'') \mid \mathsf{e}' \in \mathsf{sink}(E'), \mathsf{e}'' \in \mathsf{source}(E'')\}$. The example in Fig. 7 illustrates such a composition: we have $G = G' \triangleright G''$.

Definition 7 (Extensibility). *An axiom* $\textsc{ax}$ *satisfies extensibility if for any programs* $\mathcal{P}$ *and* $\mathcal{P}'$ *s.t.* $\mathcal{P}' \preceq \mathcal{P}$, *and* $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$ *if* $G' \models_{\mathcal{P}'} \textsc{ax}$ *and* $G'' \models_{\mathcal{P}''} \textsc{ax}$ *then* $G' \triangleright G'' \models_{\mathcal{P}} \textsc{ax}$. *An axiomatic semantics* $\mathcal{A}$ *satisfies extensibility if all axioms* $\textsc{ax} \in \mathcal{A}$ *satisfy extensibility.*

We require that the axiomatic model satisfies extensibility. We define μspecRE (RE stands for Refinable, Extensible) as the subset of μspec in which all axioms are refinable and extensible. Finite-state, bounded-history synthesis is feasible for universal axioms in μspecRE, 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 arbitiary μspec axioms. §VII-C holds our main result: we describe how axiom automata can be instantiated to get a finite-state bounded-history model for universal axioms in μspecRE.

We focus on a single universal axiom $\forall \mathtt{i}_1, \cdots, \mathtt{i}_k \phi$, but this can be easily extended to a set of axioms.

A. Axiom Automata

In what follows, we fix a (universal) axiom $\textsc{ax} = \forall \mathtt{i}_1 \cdots \forall \mathtt{i}_k \; \phi(\mathtt{i}_1, \cdots, \mathtt{i}_k)$, and let $\mathtt{I}(\textsc{ax}) = \{\mathtt{i}_1, \cdots, \mathtt{i}_k\}$, $\mathtt{E}(\textsc{ax}) = \{\mathtt{i.st} \mid \mathtt{i} \in \mathtt{I}(\textsc{ax}), \mathsf{st} \in \mathsf{Stages}\}$. 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 auxilliary definitions.

Let $\mathsf{non}\mathsf{hb}(\textsc{ax})$ denote the non-hb atoms in ϕ, i.e. instruction predicate applications and $<_r$ orderings. A *context* is an assignment (of true/false) to each atom in $\mathsf{non}\mathsf{hb}(\textsc{ax})$; $\mathsf{cxt} : \mathsf{non}\mathsf{hb}(\textsc{ax}) \rightarrow \mathbb{B}$. Each variable assignment $s : \mathtt{I}(\textsc{ax}) \rightarrow \mathbb{I}$ fixes the valuation of all $\mathsf{non}\mathsf{hb}(\textsc{ax})$ atoms (following the semantics in §II). Hence each assignment s leads to a unique context, which we denote as $\mathsf{cxt}(s)$.

We extend assignments to events and words over events. For $\mathsf{e} = \mathtt{i.st}$, we define $s(\mathsf{e}) = s(\mathtt{i}).\mathsf{st}$ and for $w \in \mathtt{E}(\textsc{ax})^*$,

$$s(w) = s(w[0]) \cdots s(w[|w| - 1]) \in \mathbb{E}^*$$

As mentioned in §III-A4, we interpret $s(w) \in \mathbb{E}^*$ as the $\mu\mathsf{hb}$ graph $w[0] \xrightarrow{\mathsf{hb}} w[1] \cdots \xrightarrow{\mathsf{hb}} w[|w| - 1]$.

Observe that once we fix the context, the validity of $\phi(\cdot)$ only depends on the value of the hb atoms in ϕ. Hence for two assignments s_1, s_2 with the same context: $\mathsf{cxt}(s_1) = \mathsf{cxt}(s_2)$, s_1 and s_2 share the same set of valid executions: $s_1(w)$ satisfies ϕ 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 $s(\mathtt{E}(\textsc{ax}))$ - 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 $\textsc{ax}$ and context cxt, we can construct a finite state automaton that recognizes acceptable orderings of $\mathtt{E}(\textsc{ax})$ (Lemma 3). The main observation behind Lemma 3 is that once the context (i.e. interpretation of the $\mathsf{non}\mathsf{hb}(\textsc{ax})$ atoms) is fixed, the allowed orderings can be represented as a language over the symbolic events $\mathtt{E}(\textsc{ax})$.

Lemma 3 (Axiom-Automata). *Given an axiom* $\textsc{ax}$ *and context* cxt, *there exists a finite-state automaton* $\mathsf{aa}(\textsc{ax}[\mathsf{cxt}])$ *over alphabet* $\mathtt{E}(\textsc{ax})$ *with language* $\{w \mid w \in \mathtt{E}(\textsc{ax})^{\mathsf{perm}}, s(w) \models \phi(\mathtt{i}_1, \cdots, \mathtt{i}_k)[s]$ *for all* s *that agree with* $\mathsf{cxt}\}$.

B. Deploying axiom automata

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

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

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

2) A basic operationalization: Lemma 3 and the concretization defined in §VII-B1 suggest an operationalization for $\textsc{ax}$. For a program $\mathcal{P}$, if a trace σ is accepted by *all* (concrete) automata $\mathsf{aa}(\textsc{ax}, \mathsf{instrsOf}(\mathcal{P}))$ then $\sigma \models \phi[s]$ holds for each assignment s, thus satisfying $\textsc{ax}$. The number of

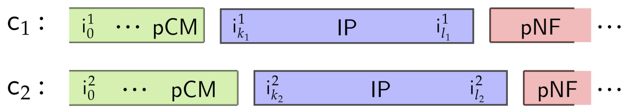


Fig. 8: Completed prefix (pCM), in-progess (IP) and not-fetched postfix (pNF) of instructions during execution.

these automata is $|\mathsf{aa}(\textsc{ax}, \mathsf{instrsOf}(\mathcal{P}))| \sim |\mathsf{instrsOf}(\mathcal{P})|^k$ for an axiom with k universally quantified variables. Since this increases with $\mathcal{P}$, the model is not finite state. Even so, this enables us to construct operational models *for a given bound on* $|\mathsf{instrsOf}(\mathcal{P})|$. We can do this even for non-universal axioms by converting existential quantifiers into finite disjunctions over $\mathsf{instrsOf}(\mathcal{P})$. 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 μspec 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 σ of a program $\mathcal{P}$ and a trace index $0 \leq j \leq |\sigma|$, let $\mathsf{CM}(j)$ and $\mathsf{NF}(j)$ be instructions which have executed all and none of their events at $\sigma[j]$ respectively. We define the following auxillary terms:

$$\begin{aligned}
\mathsf{pCM}(j) &= \{\mathsf{i} \mid \forall \mathsf{i}'.\ \mathsf{i}' \leq_r \mathsf{i} \implies \mathsf{i}' \in \mathsf{CM}(j)\} \\
\mathsf{pNF}(j) &= \{\mathsf{i} \mid \forall \mathsf{i}'.\ \mathsf{i} \leq_r \mathsf{i}' \implies \mathsf{i}' \in \mathsf{NF}(j)\} \\
\mathsf{IP}(j) &= \mathsf{instrsOf}(\mathcal{P}) \setminus (\mathsf{pCM}(j) \cup \mathsf{pNF}(j))
\end{aligned}$$

Intuitively $\mathsf{pCM}(j)$ represents the prefix-closed set of *completed* instructions, $\mathsf{pNF}(j)$ represents the postfix-closed set of *not-fetched* instructions, and $\mathsf{IP}(j)$ 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 σ, for all $0 \leq j \leq |\sigma|$, we have, $|\mathsf{IP}(j)| \leq |\mathsf{Cores}| \cdot t$.*

Active instructions Two instructions i, i' are *k-coupled* in a trace σ if they form a coupling chain of length k: i.e. there exist instructions $\mathsf{i}_1, \cdots, \mathsf{i}_{k-1}$ such that $\mathsf{coup}(\mathsf{i}, \mathsf{i}_1), \mathsf{coup}(\mathsf{i}_1, \mathsf{i}_2), \cdots, \mathsf{coup}(\mathsf{i}_{k-1}, \mathsf{i}')$. For trace σ, $0 \leq j \leq |\sigma|$ and $k \in \mathbb{N}$, we define *k-active* instructions at j, $\mathsf{AC}_k(j)$, as instructions from $\mathsf{pCM}(j) \cup \mathsf{IP}(j)$ which are k-coupled with some instruction from $\mathsf{IP}(j)$.

Intuitively, for a μspecRE 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 - $\mathsf{AC}_k(j)$ - 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 σ, for all $0 \leq j \leq |\sigma|$, we have $|\mathsf{AC}_k(j)| \leq b_k$.*

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 - 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 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.*

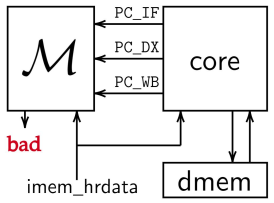


Fig. 9: Experimental setup.

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] SymbiYosys as the model-checker, with boolector [35] and abc [36] as backend solvers for BMC and PDR proof strategies respectively. Experiments

Instructions	PDR	BMC ($d = 20$)
ALU-R	1m46s	14m30s
ALU-I	2m11s	11m31s
Load+Store	2m18s	13m35s

Fig. 10: Proof runtimes for (**ax1** $\wedge$ **ax2**).

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 $|\mathsf{instrsOf}(\mathcal{P})|$.

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.

```
ax1: ∀ i1. (hb(i1.Fet,i1.DX) ∧
        hb(i1.DX,i1.WB))
ax2: ∀ i1,i2.i1<ri2 ⇒ 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 `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 `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: $\mathtt{c}_0$, $\mathtt{c}_1$, both initialized with symbolic load and store operations. We then perform verification w.r.t. the **ReadValues** (**RV**) axiom shown below. This axiom says

$\lvert I\rvert$	$\lvert AA\rvert$	BMC d	Time
4	16	12	3m10s
6	36	16	15m48s
8	64	20	1h58m

Fig. 11: Proof runtimes for the Read-Values axiom for different instruction counts ($|I|$).

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.

```
RV: ∀ i1,∃ i2,∀ i3. IsRead(i1) ⟹
   (DataInit(i1)∨(IsWrite(i2)∧
     SameAddr(i1,i2)∧hb(i2.DX,i1.DX)
     ∧ValEq(i1,i2)∧((IsWrite(i3)∧
        SameAddr(i1,i3)) ⟹
   (hb(i3.DX,i2.DX)∨hb(i1.DX,i3.DX)))))
```

This not a universal axiom, and hence Thm. 2 does not apply. However, for bounded programs we can construct $|\mathsf{instrsOf}(\mathcal{P})|^2$ 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 $\mathsf{instrsOf}(\mathcal{P})$. We perform BMC queries for programs with $|I| = |\mathsf{instrsOf}(\mathcal{P})| = 4, 6, 8$.

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 $|AA|$, 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 $|\mathsf{instrsOf}(\mathcal{P})|$.

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 **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.

```
axDep: ∀ i1, i2, (i1<ri2 ∧ Cons(i1,i2) ∧
     DepOn(i1,i2)) ⟹ hb(i1.E,i2.E)
```

We add a program counter (`pc`) to instructions and define **Cons**$(\mathrm{i}_1, \mathrm{i}_2) \equiv \mathtt{pc}(\mathrm{i}_1) + 4 = \mathtt{pc}(\mathrm{i}_2)$ and **DepOn**$(\mathrm{i}_1, \mathrm{i}_2) \equiv \mathtt{dest}(\mathrm{i}_1) = \mathtt{src1}(\mathrm{i}_2) \vee \mathtt{dest}(\mathrm{i}_1) = \mathtt{src2}(\mathrm{i}_2)$.

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

($d = 20$) 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($\mathtt{dest}(\mathsf{i}_0)$) is reset, while some instruction i_1 with $\mathtt{dest}(\mathsf{i}_0) = \mathtt{dest}(\mathsf{i}_1)$ is issued at the same cycle. A third instruction i_2 with $\mathtt{src1}(\mathsf{i}_2) = \mathtt{dest}(\mathsf{i}_0)$ 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 techinique 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: μspec specifies constraints over partial orders while LTL does so over totally ordered traces. Additionally, the elements over which constraints are enforced is also different: μspec 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 μspec 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 upto that bound, as opposed to on a per-program basis as RTLCheck does. RTL2μspec [51] aims to perform the reverse conversion: from RTL to μspec 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 μspecRE, a restricted subset of the μspec domain-specific language for axiomatic modelling. We show that the generation of an equivalent finite-state operational model is impossible for general μspec axioms, though it is feasible for universal axioms in μspecRE. 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 μspecRE (or for arbitrary μspec 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 μspec semantics (e.g., with quantitative operators) such that valid executions are guaranteed to satisfy t-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.

References

[1] Christel Baier and Joost-Pieter Katoen. Principles of model checking. 2008.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014.

[3] Yatin A. Manerkar. *Progressive Automated Formal Verification of Memory Consistency in Parallel Processors*. PhD thesis, Princeton University, Princeton, NJ, USA, 2020.

[4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994.

[5] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.

[6] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, 2011.

[7] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA*, 2016.

[8] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[9] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[10] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages*, 2:1 – 29, 2018.

[11] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[12] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646, 2014.

[13] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 463–476, 2017.

[14] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 788–801, 2018.

[15] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate : Automated exploit program generation for hardware security verification. 2018.

[16] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 26–37, 2015.

[17] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. Automated Conversion of Axiomatic to Operational Models: Theory and Practice. https://arxiv.org/abs/2208.06733, 2022.

[18] Jeremy Manson. The Java memory model. In *POPL '05*, 2005.

[19] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53:89 – 97, 2010.

[20] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL '11*, 2011.

[21] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 2005.

[22] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. Reconciling event structures with modern multiprocessors. *ArXiv*, abs/1911.06567, 2020.

[23] Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–9, 2016.

[24] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013.

[25] Stavros Aronis. Effective techniques for Stateless Model Checking. 2018.

[26] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model checking for hardware memory models. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[27] Jean Berstel. Transductions and context-free languages. In *Teubner Studienbücher : Informatik*, 1979.

[28] Jacques Sakarovitch. Elements of automata theory. 2009.

[29] Luc Maranget, Jade Alglave, Susmit Sarkar, and Peter Sewell. Litmus: Running Tests against Hardware. In *TACAS'11, 17th International Conference on Tools And Algorithms for the Construction and Analysis of Systems*, Saarbrücken, Germany, March 2011.

[30] Yatin A. Manerkar. multi-vscale. https://github.com/ymanerka/multi_vscale/tree/multicore.

[31] LGTMCU. vscale. https://github.com/LGTMCU/vscale. [Online; accessed 11-05-2021].

[32] Soham-Das-2021. Tomasulo. https://github.com/Soham-Das-2021/Tomasulo-Machine. [Online; accessed 11-05-2021].

[33] Stafford Horne. SDRAM controller. https://github.com/stffrdhrn/sdram-controller. [Online; accessed 11-05-2021].

[34] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-A Free Verilog Synthesis Suite. 2013.

[35] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[36] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/.

[37] Bruce Jacob, Spencer W. Ng, and David T. Wang. Memory systems: Cache, DRAM, disk. 2007.

[38] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10:282–312, 1988.

[39] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.

[40] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[41] Viktor Vafeiadis, Thibaut Balabonski, Soham Sundar Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[42] Conrad Watt, Christopher Pulte, Anton Podkopaev, G. Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu yu Guo. Repairing and mechanising the JavaScript relaxed memory model. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[43] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[44] J. Richard Büchi. On a decision method in restricted second order arithmetic. 1990.

[45] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 185–194, 1983.

[46] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15:519–539, 2012.

[47] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, 2002.

[48] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6:158–173, 2003.

[49] Zohar Manna and Amir Pnueli. The temporal logic of reactive and concurrent systems. In *Springer New York*, 1992.

[50] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. Reactive synthesis from signal temporal logic specifications. *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, 2015.

[51] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. *MICRO-54:*

54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021.

[52] Sanjit A. Seshia and Pramod Subramanyan. UCLID5: Integrating modeling, verification, synthesis and learning. *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10, 2018.

[53] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: Multi-modal formal modeling, verification, andsynthesis. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 538–551, Cham, 2022. Springer International Publishing.