Symbolic Execution is (Not Quite) All You Need

Version 0.1.0

Sophie Smithburg

***Abstract*—Given a programming language implementation running on a host machine and a grammar, you can use symbolic execution and formal ISA semantics as oracles to extract a formal labeled transition system that simulates the implementation's behavior.**

In plain terms: You can automatically derive executable formal specs from implementations—no one writes specs, everyone writes implementations, but you can't prove things about implementations directly. This technique lets you get the spec from the implementation (given its grammar).

***Index Terms*—symbolic execution, formal verification, labeled transition systems, semantic extraction, grammar mining, Lean 4**

I. Introduction

We can model a programming language executing on a host machine as a host labeled transition system H [1] which, after having been fed $\mathcal{I}$, the implementation, simulates G—this is to say the PL is G, its implementation is $\mathcal{I}$, and the host machine is H. If we observe a set of traces $\tau \in \mathcal{T}$ of executions of G recorded at the level of H, we can use a specific algorithm to extract an LTS G' that simulates $H_{\mathcal{I}}$.

In concrete terms, one can think of H as something like x86 or ARM, and $\mathcal{I}$ as something like CPython or V8. This matters because in practice everyone builds implementations. No one writes specs, but you can't prove things about implementations directly for the most part, and you can about specs. So in order to have an effective ratchet in the sense of the LangSec 2017 Perry Metzger talk, we want to be able to get these specs for these implementations.

To prove G' faithfully captures G, we would need G in a formalism that shares the same proof basis as G'—but in practice, this is never the case. The problem we're trying to solve practically speaking is inferring G' when no one cares to actually write out G, they're all focused on $\mathcal{I}$. So what we do instead is prove that G' simulates $H_{\mathcal{I}}$, the LTS H after executing $\mathcal{I}$, by way of quotienting over all the implementation details of H that aren't causally controllable by way of inputs to G in the form of program structure or codata.

II. Notation

- Σ is the concrete runtime space, and in our case, it is parametric over the definition of an ISA. It's basically the main memory, registers, CPU flags, any state that is documented for a particular CPU or host machine. In the more abstract view of a labeled transition system, which we focus on in this paper, Σ is simply the state space of H.
- X is computed indirectly, but can be defined directly. X is the set of all the state of the subsystem (like the programming language under analysis) that can be controlled by a program in that programming language. In the mechanization, configurations are concretely the function type Dim $\to$ Value: dimension-indexed observations of host state. The configuration space X is the image of π within this type. It is possible to extend this in the future to include input by way of any predefined input channel, but modeling codata introduces non-determinism, which is out of scope for the current paper.
- π is the mapping from Σ to X, which we get precisely by the indirect computation mechanism hinted at above.
- H is the host machine, it can be seen as an LTS or a bog standard ISA
- G is the true transition system, or notionally, what would be the specification of the programming language if the programming language were defined by the implementation? In some very real sense, a lot of programming languages are. Take, for example, C, Python, and the fact that it's been stated about the specifications in the Python enhancement proposal process and other documents in the standard that if one were to follow them, it's likely, without other guidance, they would end up with an entirely different language.
- $\mathcal{I}$ is the programming language implementation under analysis
- $H_{\mathcal{I}}$ is the LTS H after executing $\mathcal{I}$—i.e., an x86 machine after executing the CPython binary, or an ARM machine after executing a V8 implementation
- G' is the transition system we extract from traces $\tau \in \mathcal{T}$ of running $H_{\mathcal{I}}$, i.e. the host machine H or host labeled transition system without loss of generality. Additionally, G' is composed from R.
- Γ is the formal grammar for $\mathcal{I}$. We assume Γ is context-free; the approach may extend to some context-sensitive grammars, but we target the core case here.
- $\gamma \in \Gamma$ is a production in the grammar
- holes(γ) is the set of holes in production γ
- κ_h is the sentinel value for hole h in production γ
- $\mathcal{C}$ is a covering set of programs that enumerate all productions in Γ, and alternatives in each production, in a minimized way to be elaborated upon later
- $\tau \in \mathcal{T}$ is a trace, or a sequence of states and labels for the transitions between them, generated when we run the host machine H over the programming language implementation $\mathcal{I}$ and a program from $\mathcal{C}$, which implies

all the values relevant to updates (it is of course more convenient when the values for updates are provided for us; we presume this, we think without loss of generality, but perhaps that's wrong!)

- $\mathcal{T}$ is the set of all traces τ
- A **Hole-to-Hole (HTH) block** is the maximal straight-line region of execution between two consecutive holes in evaluation order—see Appendix **??** for visualizations
- L is the set of HTH labels; each $\ell = (\gamma_s, h_i, \gamma_d, h_j)$ identifies the transition from hole h_i in production γ_s to hole h_j in production γ_d, where $h_i \in \text{holes}(\gamma_s)$ and $h_j \in \text{holes}(\gamma_d)$ are consecutive in evaluation order. In the common intra-production case, $\gamma_s = \gamma_d$; cross-production transitions have $\gamma_s \neq \gamma_d$
- $\ell \in L$ is a label for a step in our transition system, identifying which HTH region the step corresponds to
- $R_\ell(x, x') := \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$ summarizes the preconditions and state transformations for a step labeled ℓ
- $R = \{R_\ell\}_{\ell \in L}$ is the family of relational summaries over all HTH labels
- $R^* = \bigcup_\ell R_\ell$ and its transitive closure
- observe : $\Sigma \to \text{Dim} \to \text{Value}$ is the observation function mapping host states to per-dimension observable values. The projection π is defined in terms of observe by restricting to tracked dimensions X
- **Observation faithfulness** (h_{faithful}): $\forall \sigma_1, \sigma_2.\ \sigma_1 \in \text{Reach}(H_\mathcal{I}) \implies (\forall d.\ \text{observe}(\sigma_1, d) = \text{observe}(\sigma_2, d)) \implies \sigma_1 = \sigma_2$. The observation function is injective on reachable states: any reachable host state is uniquely determined by its observations among all states, not just other reachable ones—σ_2 is unconstrained. This asymmetry is necessary because IsXControllable quantifies over all states in the fiber $\pi^{-1}(\pi(\sigma))$, including unreachable ones. Two unreachable states need not be distinguished from each other. This is the key hypothesis enabling co-refinement convergence (Section V).
- **X-controllability.** A label ℓ is *X-controllable* at state σ if ℓ is enabled throughout the fiber $\pi^{-1}(\pi(\sigma))$—that is, every host state with the same projected configuration can take the transition. This formalizes the concept underlying $\text{Alt}(s)$: X-controllable transitions are those whose firing depends only on X-visible state.
- $\text{Reach}(H_\mathcal{I})$ is the set of reachable states in $H_\mathcal{I}$
- $\text{BStates}(\mathcal{I}) \subseteq \text{Reach}(H_\mathcal{I})$ is the set of *all* reachable states with branching behavior—i.e., $s \in \text{BStates}(\mathcal{I})$ iff $|\text{Alt}(s)| > 1$. The branching oracle O discovers these via symbolic execution combined with formal ISA semantics (e.g., K framework reachability logic), both operating at the Σ level.
- $\text{Alt}(s)$ is the set of *all* feasible branch outcomes from state s that are reachable by modifying $\pi(s)$. This is the completeness constraint: if outcome o is achievable by changing X-state at s, then $o \in \text{Alt}(s)$. Branches depending on state outside X are either implementation-internal (irrelevant to G') or trigger π-refinement to include that state.
- $O(s)$ is the branching oracle: given $s \in \text{BStates}(\mathcal{I})$, it produces constraints over $\pi(s)$ that achieve each alternative in $\text{Alt}(s)$
- $\text{ReplayApply}(s, c) \Downarrow o$ means replaying from state s with $\pi(s)$ modified by constraint c realizes branch outcome o
- $O^*(\mathcal{T})$ is the fixpoint up to behavioral equivalence: for each branch outcome $o \in \text{Alt}(s)$ at each s along traces in $O^*(\mathcal{T})$, we require *at least one exemplar trace* realizing o. We do not generate all traces—only one representative per behavioral pattern (same control flow, same HTH structure). Additionally, π has stabilized. This suffices for completeness: every reachable behavior has an exemplar from which we extract R_ℓ.
- $G' \preceq M$ states that G' simulates M, which is to say that all behaviors of M have corresponding behaviors in G' [2], [3]

III. Main Claim

We reconstruct behavior piecewise along traces and each trace is a sound reconstruction because it almost literally replays the trace we saw. And also we use a model of the host machine that knows about branches to achieve completeness by adding new traces for all uncovered paths. If you manage soundness and completeness of reconstruction that's basically the whole game.

A. Oracles

We require two kinds of oracles: one for branching behavior and one for value transformations. Neither is counterfactual—both are instantiated by real tools.

1) Branching: The branching oracles we require are in no way counterfactual. Some aspects are partially captured by symbolic execution and then the rest of that is fully captured by formal ISA semantics. Throughout this paper, the "branching oracle O" refers to this combined realization—selective symbolic execution for path feasibility plus formal ISA semantics (e.g., the K framework [4]) for reachability—not an additional assumption beyond symex. We use O as notational shorthand.

a) Completeness.: For all $\mathcal{I}$-relevant branch states and all feasible outcomes, the oracle can produce a constraint that achieves that outcome:

$$\forall s \in \text{BStates}(\mathcal{I}).\ \forall o \in \text{Alt}(s).\ \exists c \in O(s).\ \text{ReplayApply}(s, c) \Downarrow o$$

2) Value Transformation: What we need for value transformation inference is provided fully by symbolic execution engines. When symbolically executing a basic block, the engine produces symbolic expressions showing how input values map to output values—this directly gives us the $\text{Update}(x, x')$ component of each R_ℓ [5], [6].

B. Proof Structure

1) **Given**: H (host), $\mathcal{I}$ (implementation), Γ (grammar)
2) **Given**: O (branching oracle)—instantiated by SSE or formal ISA semantics

3) **Define**: $\mathcal{T}_0$ = initial traces from covering set $\mathcal{C}$
4) **Define**: $O^*(\mathcal{T}_0)$ = fixpoint (apply O until no new traces)
5) **Assume**: Symbolic execution recovers Guard $\wedge$ Update from HTH regions
6) **Construct**: G' from $\{R_\ell \mid \ell \text{ derived from } O^*(\mathcal{T}_0)\}$
7) **Prove**: $G' \preceq H_\mathcal{I}$

C. Soundness and Completeness

a) Soundness.: Each R_ℓ is computed as the relational transformer associated with a hole-to-hole basic block. Our relational transformer is as sound as the symbolic execution engine we use to compute said transformer.

b) Completeness.: This follows basically from the definition of the branching oracle and the fixpoint. Since by definition the branching oracle can detect all branching behavior at the level of H, were there to be any state reachable in $H_\mathcal{I}$ that we did not find at least one exemplar trace for, then it's a contradiction with the assumption that the oracle is complete. This is conditional on oracle completeness (symex being biconditional with $H_\mathcal{I}$.step); given this, the bisimulation Corollary (Section V-E, `extraction_bisimulation`) formally establishes both simulation directions, ensuring no reachable behavior is missed.

D. Scope of This Paper

This current paper attempts to prove that it is possible to do such a reconstruction. We leave it for future work to fully implement such an algorithm and prove it correct. We don't consider it necessary to prove that the oracles exist because they do in the form of formal specifications of ISAs and symbolic execution engines. Our intent is simply to prove that, given such tools, it should be possible to extract semantics from the implementations and an algorithm corresponding to this sketch.

We're only focused on ISAs, like Industrial ISAs, running industrial programming languages. We are not yet aiming to tackle them in their full generality, but rather to model as large of a slice of an industrial implementation as is possible. To this end, we make some radical choices in terms of scope to make proving any result possible, knowing that we must return to each of these in future work.

Out of scope for this paper are:

- non-determinism
- concurrency
- lazy evaluation
- typing or any static analysis or codata

Each of these scope limitations would likely warrant their own follow-up paper in themselves. All of that being said, we think that this is still useful work, as the core semantics of a large class of industrially interesting programming languages can be captured this way.

IV. Methodology

We use sentinels κ_h in templates $P \in \mathcal{C}$ to determine the correspondence from a program with a particular AST construct executing and the resulting trace—particularly the mapping from sentinel holes to their corresponding values in the trace. We use this both to learn a mapping from syntactic representation to runtime representation, as well as to delineate the bounds of hole-to-hole basic blocks.

A. Structured Programming and Lexical Scope

[NOTE: Sophie flagged this may need to move earlier]

To explain what a hole-to-hole basic block is, we must first introduce the requirement of lexical scope in structured programming languages. Structured programming is necessary because it gives us the property that jumps are always lexically bounded with respect to the AST. For κ_{h_1} to κ_{h_2}, that basic block, we identify the points in the trace that correspond to that. From those points in the trace, we can extract information like memory regions to feed to a selective symbolic execution engine to selectively symbolically execute just that region between those two holes.

B. Learning the Configuration

We are also using this to learn π. The idea is that we learn what our configuration should look like. Our configuration is defined by the bindings from AST paths to runtime values, so if something isn't AST-bindable or AST-referable, then it can't end up in our projection π.

Differential causality testing. To learn π concretely, we run paired experiments. For each hole h in template P, we execute P with two different sentinel values: $P[h \leftarrow \kappa_1]$ and $P[h \leftarrow \kappa_2]$. Comparing the resulting traces τ_1 and τ_2, we identify the positions where they differ. These differing positions represent state that is *causally influenced* by the value at hole h—and therefore must be included in X. This is analogous to dynamic invariant discovery [7], but over causal influence rather than likely invariants.

The key insight is that X is exactly the *transitive closure of AST-bound state*. Nothing enters X without first being directly bound to an AST position (by determinism—all state changes trace back to program structure). A variable's value is in X because it was assigned from an expression; that expression's value is in X because it was computed from subexpressions bound to AST nodes; and so on. The differential test reveals this transitive closure: if changing a sentinel at hole h eventually affects some state S in the trace, then S is transitively reachable from h's AST binding, and belongs in X.

C. Control Flow and Value Transformation

We extract control flow and value transformation along this same part of the technique. The selective symbolic execution gives us path constraints. Say we're doing the if-then-else situation where we are going from `cond` to the then-branch. We'll have a path constraint induced by this that `cond` must evaluate to something truthy. We can flip that path constraint, and that gives us the other thing we have to explore. We would have already gotten that by way of our covering set in the case of truthy and falsy—at least for literal `true` and `false` we would have. What's key is that we get an SMT formulation in terms of things we have determined bindings for in terms of

our AST—we've extracted the guard conditions for our labeled transition system.

For value inference: since these hole-to-hole blocks do not involve branching, we can extract the relational transformer along this basic block. We use the symbolic statement of inputs in the configuration mapping to symbolic expressions in the output. We can use those, plus the mapping from syntactic literal to runtime type, to define the value transformation aspect of semantics.

D. Bootstrapping and Co-Refinement

The boundary detection problem. Finding HTH regions requires knowing where each hole's computation "happens" in the trace. But we cannot simply look for where sentinel *values* appear—values can be inlined, constant-folded, or otherwise transformed by the implementation. We need to detect where *computation* happens, not just where values appear.

Moreover, we cannot use syntactic heuristics like "this AST node is named `if`" or "this node has two children." The entire point is to infer semantics from implementation behavior. If we hardcode syntactic patterns, we assume what we are trying to learn.

Sentinel expressions. Instead of sentinel values, we use sentinel *expressions* that force observable computation—e.g., `(x := sentinel; sentinel)` where the assignment leaves a trace signature. When the assignment appears in the trace, we know that hole was evaluated. But to design such expressions, we must know which operations have observable sequencing.

Learning algebraic laws from R_ℓ. Here is the key insight: R_ℓ is an SMT formula produced by symbolic execution. SMT formulas use SMT-LIB operations: `bvadd`, `bvmul`, `store`, etc. These operations have *precisely defined algebraic laws*—this is the SMT-LIB standard. We do not pattern-match against heuristics; we read exact algebraic properties from the formula structure to identify what *weakens* ordering detection:

- R_ℓ contains $\texttt{bvadd}(x, y) \Rightarrow$ commutative $\Rightarrow$ *cannot* use this to detect evaluation order
- R_ℓ contains $\texttt{bvsub}(x, y) \Rightarrow$ non-commutative $\Rightarrow$ *can* use this to detect order
- R_ℓ contains `store` $\Rightarrow$ has observable side effects $\Rightarrow$ *can* use this to detect order

Sequencing analysis. Commutative operations cannot detect evaluation order: if $a + b = b + a$, we cannot tell from the result whether a or b was evaluated first. Non-commutative operations and side effects *can* detect order. By reading the algebraic laws from R_ℓ, we identify which operations are safe for sequencing detection.

The bootstrapping loop. This creates a productive cycle:

1) Start with rough HTH regions from initial trace differences and observable side effects
2) Extract R_ℓ via selective symex on each region
3) Read algebraic laws from R_ℓ (exact, from SMT-LIB operations)
4) Use laws to refine sequencing analysis—design better sentinel expressions
5) Get more precise HTH boundaries
6) Extract more precise R_ℓ
7) Repeat until fixpoint

Convergence. This process terminates because: (1) the grammar Γ is finite, so there are finitely many operations to analyze; (2) algebraic law knowledge is monotonic—we only learn new laws, never unlearn; (3) HTH precision increases or stays the same with each iteration. The fixpoint is reached when no new algebraic knowledge yields more precise boundaries.

Three-way co-refinement. The full technique involves co-refinement across three dimensions:

1) **Configuration refinement (π):** Learning what state is program-relevant by observing causal influence of sentinels
2) **Region refinement (HTH):** Learning computation boundaries using sequencing-sensitive expressions, identified by their non-commutative semantics
3) **Semantic refinement (R_ℓ):** Extracting relational transformers via selective symex, which reveals algebraic properties that inform sequencing detection

These three refine together: better $\pi \rightarrow$ better HTH detection $\rightarrow$ better $R_\ell \rightarrow$ better algebraic knowledge $\rightarrow$ better HTH $\rightarrow \ldots$ until all three stabilize together with O^*.

E. Soundness and Completeness of the Technique

This gives us a representation of individual paths that is as sound as either our symex engine and/or our formal semantics, to the extent we depend on either of those oracles and/or both. It is complete as both of those oracles as well. Since it's lexical scope, we don't have to do any scope inference. [That's the whole technique.]

V. Proof

This section proves a conditional result: given a sound oracle for symbolic execution (subsuming grammar coverage, control-flow branching, and value transformation—see Section V-A), the reconstructed semantics G' simulates $H_{\mathcal{I}}$ within the constrained semantic domain defined in the Scope section. With the additional hypothesis that the oracle is also complete, we obtain bisimulation (Corollary V).

A. Oracle Decomposition

We decompose the reconstruction's correctness into three oracle dependencies:

1) **Grammar oracle** (Γ): Provides full syntactic coverage of the language fragment. This may be a known grammar or one inferred via grammar-mining techniques [8]. The grammar oracle delimits the syntax space but is not a source of semantic uncertainty.
2) **Branching oracle** (O, Section III-A): The critical completeness dependency. We require *local completeness* over branching behavior relevant to each reconstructed HTH region. The combined symex+ISA oracle (Section III-A) serves as a branching discriminator, ensuring all feasible successor regions for a given syntactic construct are discovered.

3) **Value-transformation oracle**: Provides relational transformers $R_\ell(x, x')$ for basic blocks. May be instantiated by symbolic execution, relational symbolic execution, or trusted formal ISA semantics. This oracle primarily affects soundness: incorrect transformers yield unusable semantics, while partial transformers under-approximate behavior.

In the Lean mechanization, the branching and value-transformation oracles are unified into a single parameter symex : $L \to \Sigma \to \Sigma \to \text{Prop}$, which soundly approximates $H_\mathcal{I}$'s transition relation. The existential projection of symex through π captures both guard conditions (via the domain of the projected relation) and value updates (via its codomain). The grammar oracle is a structural precondition (`GrammarConformant`), not a runtime parameter. The three-way decomposition above is conceptual; the mechanization shows that a single sound symex oracle suffices.

B. Setup

Fix a host machine H, implementation $\mathcal{I}$, and grammar Γ. Let:

- $\mathcal{C}$ be a covering set of programs that exercises every production in Γ
- $\mathcal{T}_0 = \{\tau_P \mid P \in \mathcal{C}\}$ be the initial traces from executing $\mathcal{C}$ on $H_\mathcal{I}$
- O be the symex+ISA oracle (Section III-A) satisfying the completeness axiom
- $O^*(\mathcal{T}_0)$ be the fixpoint of applying O to traces until no new traces are discovered and π has stabilized

Labeling HTH regions (precondition). HTH labels are assigned by construction in the template setting, not inferred from arbitrary program traces: each template exercises a single production with sentinels at holes, and the trace segment between consecutive sentinel evaluations defines the HTH block whose label is determined by template structure and observed evaluation order. Concretely, each template $P \in \mathcal{C}$ is derived from some production $\gamma \in \Gamma$. The trace region between the computation of hole h_i and hole h_j—where h_i and h_j are consecutive in that trace's evaluation order—constitutes an HTH region, and we assign it the label $\ell = (\gamma_s, h_i, \gamma_d, h_j)$ where γ_s is the production containing h_i and γ_d is the production containing h_j (typically $\gamma_s = \gamma_d$). For traces generated by exploring branches, different control flow paths may reach different holes, yielding different labels. The mechanism for detecting these boundaries—using sentinel expressions refined via algebraic laws—is described in Section IV-D.[1]

For each trace $\tau \in O^*(\mathcal{T}_0)$, the value-transformation oracle produces relational summaries $R_\ell(x, x') := \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$ for each HTH region labeled ℓ within τ.

Define $G' := \langle X, L, \{R_\ell\}_{\ell \in L}, s_0 \rangle$ where:

- X is the learned configuration space (the image of π)
- L is the set of HTH labels (see Notation)
- $s_0 = \pi(\sigma_0)$ for some initial $H_\mathcal{I}$ state σ_0

[1] In lazy languages, demand patterns determine which holes are evaluated and when, introducing completeness concerns that sentinels and symbolic execution cannot address—one would need to instrument demand propagation itself. This is why we restrict to eager evaluation.

C. Co-Refinement Fixpoint

Definition (Co-Refinement Fixpoint). The triple $(H_\mathcal{I}, \pi, R)$ satisfies the *co-refinement fixpoint condition* if:

1) **Oracle soundness**: every concrete transition is captured—$\forall \sigma, \sigma', \ell.\ \sigma \xrightarrow{\ell} \sigma' \implies R_\ell(\pi(\sigma), \pi(\sigma'))$
2) **Non-controllable preservation**: non-X-controllable transitions from reachable states preserve the projection—$\forall \sigma, \sigma', \ell.\ \sigma \in \text{Reach}(H_\mathcal{I}) \wedge \sigma \xrightarrow{\ell} \sigma' \wedge \neg\text{IsXControllable}(\sigma, \ell) \implies \pi(\sigma) = \pi(\sigma')$

Together these ensure the extracted LTS faithfully represents $H_\mathcal{I}$'s behavior at the granularity captured by π: observable transitions are in R's domain, and unobservable transitions don't change the projected state.

D. Extraction Construction

We construct a concrete co-refinement process whose fixpoint yields π and R satisfying the conditions above. The construction uses three components:

- **Projection.** For a tracked dimension set $X \subseteq \text{Dim}$, define $\pi_X(\sigma)(d) = \text{observe}(\sigma, d)$ if $d \in X$, else a default value. Two states have the same projected configuration iff they agree on all tracked dimensions. The configuration type is $\text{Dim} \to \text{Value}$, so configurations are equal iff they agree on every dimension—tracked dimensions carry observed state, while untracked dimensions are padded with a fixed default.
- **Oracle.** Given a symbolic execution oracle symex : $L \to \Sigma \to \Sigma \to \text{Prop}$ that soundly approximates $H_\mathcal{I}$'s transitions ($\forall \sigma, \sigma', \ell.\ \sigma \xrightarrow{\ell} \sigma' \implies \text{symex}(\ell, \sigma, \sigma')$), define $R_\ell^X(x, x') \iff \exists \sigma, \sigma'.\ \pi_X(\sigma) = x \wedge \text{symex}(\ell, \sigma, \sigma') \wedge \pi_X(\sigma') = x'$. Soundness follows from the symex oracle's soundness: every concrete transition $\sigma \xrightarrow{\ell} \sigma'$ yields $\text{symex}(\ell, \sigma, \sigma')$, providing the existential witness. In practice, the symex oracle is instantiated by symbolic execution of HTH regions in templates from the covering set, producing $\text{Guard}_\ell \wedge \text{Update}_\ell$ (Section IV-C). Note that this existential projection is the mechanized form of the $R_\ell(x, x') := \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$ from Section II: when symex factors as a path condition plus substitution (i.e., $\text{symex}(\ell, \sigma, \sigma') \iff \text{PC}_\ell(\sigma) \wedge \text{Sub}_\ell(\sigma) = \sigma'$, as in the ICTAC setting), the existential collapses to the guard-and-update decomposition.
- **Refinement step.** Given X, compute $X' = X \cup \{d \mid \exists \sigma, \sigma_2, \ell.\ \sigma \in \text{Reach}(H_\mathcal{I}) \wedge (\exists \sigma'.\ \sigma \xrightarrow{\ell} \sigma') \wedge \pi_X(\sigma_2) = \pi_X(\sigma) \wedge (\nexists \sigma_2'.\ \sigma_2 \xrightarrow{\ell} \sigma_2') \wedge \text{observe}(\sigma, d) \neq \text{observe}(\sigma_2, d)\}$. That is: add dimension d whenever two states with the same projection differ at d and have different transition availability—a branch divergence witness.

Bridge to differential causality. The refinement step's witnesses are exactly the branch divergence witnesses that

differential causality testing detects (Section IV-B): two states at a branch point with the same projection but different observations at dimension d, where one can take a transition and the other cannot. This justifies using differential causality testing as the concrete mechanism for discovering refinement dimensions.

Convergence. Since Dim is finite and each refinement step is inflationary ($X \subseteq X'$), the process stabilizes in at most $|\text{Dim}|$ steps by monotone stabilization of finite sets. The oracle O operates at the Σ level (full host state), so it is never blocked waiting for π to stabilize—symbolic execution sees all concrete state, not just the projection. The π refinement determines what we *include in* G', not what O can discover.

Fixpoint property. At the fixpoint $X^* = \text{refineStep}(X^*)$, no non-$X^*$-controllable transitions exist among reachable states. Suppose for contradiction that $\sigma \in \text{Reach}(H_{\mathcal{I}})$ can take transition ℓ (with witness σ') but some σ_2 with $\pi_{X^*}(\sigma_2) = \pi_{X^*}(\sigma)$ cannot. We show $\sigma = \sigma_2$, contradicting the assumption. Fix an arbitrary dimension d:

1) **Case** $d \in X^*$: By projection equality, $\pi_{X^*}(\sigma_2) = \pi_{X^*}(\sigma)$. Evaluating at $d \in X^*$ gives $\text{observe}(\sigma, d) = \text{observe}(\sigma_2, d)$.
2) **Case** $d \notin X^*$: If $\text{observe}(\sigma, d) \neq \text{observe}(\sigma_2, d)$, then $(\sigma, \sigma_2, \ell, d)$ is a valid refinement witness—so $d \in \text{refineStep}(X^*)$. But $\text{refineStep}(X^*) = X^*$, so $d \in X^*$, contradicting $d \notin X^*$. Therefore $\text{observe}(\sigma, d) = \text{observe}(\sigma_2, d)$.

Since $\text{observe}(\sigma, d) = \text{observe}(\sigma_2, d)$ for all d, observation faithfulness (h_{faithful}, using $\sigma \in \text{Reach}(H_{\mathcal{I}})$) gives $\sigma_2 = \sigma$. But σ can take ℓ, so σ_2 can too—contradiction. (Mechanized: `ExtractionPossibility.lean`, lines 222–249.)

E. Main Theorem

Theorem (`extraction_pipeline`). Given:

1) A grammar-conformant implementation $H_{\mathcal{I}}$ with grammar Γ
2) An observation function observe : HostState $\to$ Dim $\to$ Value over a finite dimension set[2]
3) Observation faithfulness (h_{faithful}): $\forall \sigma_1, \sigma_2.\ \sigma_1 \in \text{Reach}(H_{\mathcal{I}}) \implies (\forall d.\ \text{observe}(\sigma_1, d) = \text{observe}(\sigma_2, d)) \to \sigma_1 = \sigma_2$
4) A sound symbolic execution oracle symex : $L \to \Sigma \to \Sigma \to$ Prop satisfying $\forall \sigma, \sigma', \ell.\ \sigma \xrightarrow{\ell} \sigma' \implies \text{symex}(\ell, \sigma, \sigma')$

there exist π and R satisfying the co-refinement fixpoint condition, and hence $G' \preceq H_{\mathcal{I}}$.

Note. The symex oracle is instantiated in practice by symbolic execution of HTH regions in templates from the covering set (Section IV-C), producing the $\text{Guard}_\ell \wedge \text{Update}_\ell$ decomposition. The testing infrastructure—covering sets, differential causality testing, reachability oracle—discovers which dimensions to track (X) and which addresses to symbolize; the symex oracle captures how transitions transform state.

Composition chain. The hypothesis $h_{\text{symex\_sound}}$ is discharged by composing three stages: (1) the covering set algorithm produces one template per grammar rule (`coveringSet_adequate`); (2) for each HTH label ℓ, symbolic execution of the corresponding template region yields a $\text{Guard}_\ell \wedge \text{Update}_\ell$ summary, as established by ICTAC's `trace_correspondence` or Lucanu et al.'s generic symbolic execution framework [9]; (3) per-label soundness lifts to $h_{\text{symex\_sound}}$ by universal quantification over labels. For each label ℓ, we assume the symex engine's Guard/Update summary is *parametric* in the symbolic inputs corresponding to holes—that is, the summary is sound for all concrete instances of ℓ, not just the template program used to elicit it.

Proof. The proof proceeds in two parts.

Part 1: Identifying projection dimensions. Differential causality testing (Section IV-B) identifies the dimensions that must be tracked in π. For each template $P \in \mathcal{C}$ and each hole h, paired executions with distinct sentinel values reveal which output dimensions are causally influenced by the value at h. The reachability oracle witnesses these causal connections: a dimension differs in the output iff the oracle witnesses causal influence from the sentinel injection point. These identified dimensions, together with branch divergence witnesses from control-flow exploration, provide the inputs to the refinement step of the co-refinement process.

Part 2: Co-refinement yields simulation. Apply the extraction construction (Section V-D) starting from $X_0 = \emptyset$. By convergence, the process reaches a fixpoint X^* with projection $\pi = \pi_{X^*}$ and oracle $R = R^{X^*}$. At the fixpoint:

- *Oracle soundness* follows from $h_{\text{symex\_sound}}$: every concrete transition $\sigma \xrightarrow{\ell} \sigma'$ gives $\text{symex}(\ell, \sigma, \sigma')$ by soundness, which with $\pi_{X^*}(\sigma)$ and $\pi_{X^*}(\sigma')$ provides the existential witness for R^{X^*}.
- *Non-controllable preservation* holds vacuously: at the fixpoint, h_{faithful} implies no non-X^*-controllable transitions exist among reachable states (see Fixpoint property above).

Therefore $(H_{\mathcal{I}}, \pi, R)$ satisfies the co-refinement fixpoint condition, and $G' \preceq H_{\mathcal{I}}$ follows. □

Corollary (`extraction_bisimulation`). With the additional hypothesis that symex is *complete*—every symex claim corresponds to a real step ($\forall \sigma, \sigma', \ell.\ \text{symex}(\ell, \sigma, \sigma') \implies \sigma \xrightarrow{\ell} \sigma'$), making symex biconditional with $H_{\mathcal{I}}$.step—the extraction pipeline yields bisimulation among reachable states: both $G' \preceq H_{\mathcal{I}}$ and $H_{\mathcal{I}} \preceq G'$ hold with witness relations restricted to $\text{Reach}(H_{\mathcal{I}})$. Oracle soundness alone suffices for simulation (Main Theorem); oracle completeness is additionally required for the reverse direction.

[2]The Lean mechanization additionally requires decidable equality on dimensions (`[DecidableEq Dim]`), finiteness of the dimension type (`[Fintype Dim]`), and an inhabited value type (`[Inhabited Value]`) for the default padding in the projection outside tracked dimensions. These are mild technical conditions satisfied by any concrete ISA instantiation (e.g., finite register indices and machine words).

F. Remarks

On oracle instantiation. None of the hypotheses are counterfactual. The *grammar* is either given (language specification) or inferred via grammar-mining techniques. The *branching oracle* is instantiated by selective symbolic execution engines (S2E, angr) and formal ISA semantics (Sail for ARM/RISC-V) [4], [10]. The *value-transformation oracle* is similarly instantiated by symbolic execution's production of path conditions and symbolic stores. *Observation faithfulness* (h_{faithful}) requires that each reachable state be distinguishable by observation from *all* states (not just other reachable ones); two unreachable states need not be distinguished from each other. This holds whenever the observation function includes all ISA-documented state (registers, memory, flags) that a program can reach.

On the conditional nature. The nontrivial content of this result lies in making the reconstruction mechanism explicit, not in claiming stronger guarantees than the oracles permit. Soundness holds insofar as the value-transformation oracle is sound; completeness holds insofar as the branching oracle enumerates all feasible control-flow alternatives.

On the bisimulation construction. The simulation proof (`extraction_possible`) cannot be directly extended to bisimulation by "just adding $h_{\text{symex\_complete}}$." The obstacle is the reverse direction: given an oracle claim $R(\ell, \pi(\sigma), x')$, the existential witness σ_0 with $\pi(\sigma_0) = \pi(\sigma)$ need not equal σ—two distinct host states can share a projection but differ on untracked dimensions, producing different transition behavior. Completeness requires that the witness's step *is* the query state's step, which fails when π is not injective on reachable states.

The mechanized proof (`extraction_bisimulation`) uses an independent construction with three differences from the simulation proof:

1) **Different refinement step.** Instead of tracking branch divergence witnesses (where a reachable state can take a transition but some other state with the same projection cannot), the bisimulation refinement tracks *observation disagreements among reachable states*: add dimension d whenever two *reachable* states σ_1, σ_2 with the same projection differ at d. Both states must be reachable, not just one.
2) **Reachability-restricted oracle.** The projected oracle additionally requires the existential witness σ to be reachable: $R_\ell(x, x') \iff \exists \sigma, \sigma'.\ \sigma \in \text{Reach}(H_{\mathcal{I}}) \wedge \pi(\sigma) = x \wedge \text{symex}(\ell, \sigma, \sigma') \wedge \pi(\sigma') = x'$. This ensures that oracle claims are grounded in reachable behavior.
3) **π-injectivity at fixpoint.** At fixpoint, any two reachable states with the same projection agree on all dimensions: tracked dimensions agree by projection equality; untracked dimensions agree because disagreement would trigger refinement, contradicting the fixpoint. By h_{faithful}, the states are equal. This makes π injective on $\text{Reach}(H_{\mathcal{I}})$.

Completeness from injectivity. Given a reachable σ and an oracle claim $R(\ell, \pi(\sigma), x')$ witnessed by some reachable σ_0 with $\pi(\sigma_0) = \pi(\sigma)$, injectivity gives $\sigma_0 = \sigma$. So the witness's symex claim is about σ itself, and $h_{\text{symex\_complete}}$ gives a real step $\sigma \xrightarrow{\ell} \sigma'$. No observation-determinism or functional-substitution hypothesis is needed—the reachability refinement does all the work.

The ICTAC case and tractability. In the ICTAC setting ($\pi = \text{id}$), bisimulation follows directly from trace correspondence without the reachability refinement machinery (`bisimulation_of_TraceCorrespondence_id`). The reachability restriction in the general case is natural: it excludes only states unreachable by any program execution, which are irrelevant to language semantics. For bounded HTH fragments—straight-line regions between sentinel boundaries with minimal path explosion—symex completeness is tractable when backed by formal ISA semantics (e.g., K framework, Sail), making bisimulation a realistic outcome for the intended setting rather than a theoretical curiosity.

On grammar conformance. Grammar conformance (`GrammarConformant`) is a precondition of the theorem, not something the proof constructs. It asserts that the implementation's reachable transitions are structured by the grammar: every reachable step is labeled by rules from Γ. This is a joint condition on Γ and $H_{\mathcal{I}}$—having the right grammar is necessary but not sufficient; the implementation must actually conform to it. In practice, conformance is established by grammar mining paired with trace validation against $H_{\mathcal{I}}$.

On tractability. The co-refinement loop terminates in at most $|\text{Dim}|$ iterations (mechanized as `inflationary_stabilizes_bound`). Each iteration examines all dimensions, so the algorithm's control structure is quadratic in $|\text{Dim}|$. The dominant cost is the oracle (symbolic execution) at each step—this cost is inherited from the symex engine and is feasible for bounded HTH fragments under the scope constraints (eager evaluation, no concurrency, no nondeterminism).

On generality. The main theorem of this paper is proved for the specific setting of LTS extraction from an ISA implementation. However, the core argument depends on surprisingly little structure. We mechanize (`Learnability.lean`, independent of all other project modules) three sufficient *learnability preconditions* under which *any* system expressible as State $\to$ Label $\to$ State $\to$ Prop admits a faithful projected model via iterative refinement:

1) **Finiteness**: the observation space (dimensions) is finite, hence enumerable.
2) **Identifiability**: observations distinguish relevant states—concretely, a relevant state must be distinguishable from *all* states (not just other relevant ones), since controllability quantifies over the full state space.
3) **Extractibility**: a sound oracle witnesses behavior—every real behavior has an oracle witness.

Two further properties—*enumerability* (dimensions can be iterated) and *separability* (at the fixpoint, the projection captures all relevant distinctions)—follow from finiteness and the refinement construction respectively, rather than being assumed.

Under these conditions, `extraction_exists` proves the existence of a dimension set X^* at which the projected oracle is sound for all relevant behaviors and every relevant state's behavior availability is controllable (same projection implies same behavioral capabilities). With a complete oracle, `exact_extraction` strengthens this to injectivity on relevant states, from which the full bisimulation follows. The proof strategy—contradiction plus pigeonhole on a monotone chain in Finset(Dim)—is the same one used in the main paper, abstracted away from LTS-specific vocabulary.

This paper validates the framework for the LTS case only. Instantiating the preconditions for non-LTS domains—particularly faithfulness with unconstrained s_2 and constructing a sound oracle—is non-trivial, and per-domain tractability analysis remains an open problem (see the tractability remark above). That said, the relation State $\to$ Label $\to$ State $\to$ Prop is in principle general enough to encode type systems, parsers, effect systems, and other semantic domains, though we have not validated the preconditions for any of these.

References

[1] T. A. Henzinger, R. Majumdar, and J.-F. Raskin, "A classification of symbolic transition systems," in *Proceedings of STACS 2000*, ser. Lecture Notes in Computer Science, vol. 1770. Springer, 2000, pp. 13–34.

[2] R. J. van Glabbeek, "The linear time – branching time spectrum," in *CONCUR '90: Theories of Concurrency: Unification and Extension*, ser. Lecture Notes in Computer Science, vol. 458. Springer, 1990, pp. 278–297.

[3] ——, "The linear time – branching time spectrum II: The semantics of sequential systems with silent moves," in *CONCUR '93: 4th International Conference on Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 715. Springer, 1993, pp. 66–81.

[4] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[5] E. Voogd, Å. A. A. Kløvstad, E. B. Johnsen, and A. Wasowski, "Compositional symbolic execution semantics," *Theoretical Computer Science*, vol. 1044, p. 115263, 2025.

[6] A. Lööw, S. H. Park, D. Nantes-Sobrinho, S.-É. Ayoun, O. Sjöstedt, and P. Gardner, "Compositional symbolic execution for the next 700 memory models," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2025.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[8] L. Bettscheider and A. Zeller, "Look ma, no input samples! Mining input grammars from code with symbolic parsing," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM, 2024, pp. 333–337.

[9] D. Lucanu, V. Rusu, and A. Arusoaie, "A generic framework for symbolic execution: A coinductive approach," *Journal of Symbolic Computation*, vol. 80, pp. 125–163, 2017.

[10] J. Craaijo, F. Verbeek, and B. Ravindran, "libLISA: Instruction discovery and analysis on x86-64," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1–29, 2024.