RE3: Finding Refinement Relations with Relational Mapping Abstraction

You Li*, Guannan Zhao*, Yunqi He, and Hai Zhou
Northwestern University, Evanston, USA
{you.li, gnzhao, yunqi.he}@u.northwestern.edu, haizhou@northwestern.edu

***Abstract*—A refinement relation captures the state equivalence between two sequential circuits. It finds applications in various tasks of VLSI design automation, including regression verification, behavioral model synthesis, assertion synthesis, and design space exploration. However, manually constructing a refinement relation requires an engineer to have both domain knowledge and expertise in formal methods, which is especially challenging for complex designs after significant transformations. This paper presents a rigorous and efficient sequential equivalence checking algorithm for non-cycle-accurate designs. The algorithm can automatically find a concise and human-comprehensible refinement relation between two designs, helping engineers understand the essence of design transformations. We demonstrate the usefulness and efficiency of the proposed algorithm with experiments and case studies. In particular, we showcase how refinement relations can facilitate error detection and correction for LLM-generated RTL designs.**

I. Introduction

Design engineers and optimization tools are performing even more aggressive sequential transformations to meet the growing demands of high performance and low power consumption. Examples of sequential transformations include retiming, pipelining, pre-computation, resource reallocation, clock gating, and memory partitioning. Recently, adaptive pipelining and latency-insensitive design methodologies have been proposed to further improve the throughput of high-performance integrated circuits. These techniques can cause irregular data dependencies and dynamic latencies [1] and are prone to error.

Sequential equivalence checking is the key enabler of these transformations. It assures that the two designs before and after a series of transformations are functionally equivalent across clock cycles. Traditional sequential equivalence checking algorithms require that the two designs are cycle-accurate and have a static one-to-one latch mapping. However, these assumptions are no longer valid after radical sequential transformations. On the other hand, non-cycle-accurate sequential equivalence checking algorithms [2], [3] can tolerate timing differences between the two designs. Nevertheless, they encounter scalability issues on large industrial designs because their search space is a product of the state space of the two designs and the space of all possible alignment patterns.

A refinement relation captures the state equivalence between two functionally equivalent designs. It has wide applications in various stages of electronic design automation. For instance, regression verification leverages refinement relations to determine whether a sequence of successively modified designs are functionally equivalent [4], [5]. Behavioral model synthesis takes a refinement relation to generate a system-level reference model using program synthesis techniques [6], [7]. Refinement relation is also crucial for formal-guided design space exploration and root cause analysis [8]. Notably, the emerging Instruction-level Abstraction (ILA) framework relies on refinement relations for various tasks, including property checking and assertion synthesis [9], [10]. Manually constructing a refinement relation requires both domain knowledge and expertise in formal methods. It is notoriously difficult for humans to write a correct refinement relation for complex designs after significant transformations [6], [9].

This paper presents RE3, an efficient symbolic model checking algorithm for non-cycle-accurate sequential equivalence checking. RE3 employs an orchestration of techniques to automatically extract a concise and human-comprehensible refinement relation from two equivalent hardware designs. Specifically, it *i)* reduces the problem complexity by dividing the equivalence checking problem into two tasks: synthesizing a scheduling function and finding a corresponding inductive invariant; *ii)* steers the searching process towards a simple refinement relation by controlling the granularity of the scheduling function; *iii)* captures the dynamic latch mapping between two hardware designs by constructing a dedicated relational mapping abstraction domain; *iv)* filters out bad states with a lightweight simulation-based self-alignment mechanism; and *v)* ensures the incrementality of the model checking process and maintains the compatibility with state-of-the-art model checking algorithms. With all those techniques, RE3 addresses the challenges in non-cycle-accurate sequential equivalence checking and generates high-quality refinement relations to facilitate various downstream tasks.

This paper makes the following key contributions:

- It proposes a rigorous and efficient non-cycle-accurate sequential equivalence checking algorithm;
- It develops a novel abstraction scheme dedicated to hardware equivalence checking;
- It presents the first approach to automatically synthesize a concise and human-comprehensible refinement relation for hardware designs;
- It demonstrates the roles and potentials of refinement relations in conventional and AI-based VLSI design and verification flows.

II. Problem Definition and Analysis

A. Preliminaries

We consider standard first-order logic. A *term* is a variable or a *function* symbol. The terms with non-Boolean values are also referred to as *words*. A *predicate* represents a relation over a set of terms and evaluates to a Boolean value. An *atom* is either a predicate symbol or a Boolean variable. A *formula* is built from atoms with propositional logic. A *literal* is either an atom or a negated atom. A *clause* is a disjunction of literals, whereas a *cube* is a conjunction of literals.

A transition system M is defined as a tuple $\langle X, I, T\rangle$, where X denotes the set of *state variables*, X' denotes the corresponding set of next-state variables, $I(X)$ is a formula representing the *initial condition*, and $T(X, X')$ is a formula representing the *transition relation*. It is a common practice to model input variables as additional state variables [11]–[13] such that $X = X_{state} \cup X_{in}$. The next-state input variables X'_{in} are either unconstrained or controlled by an external specification. A state $s \in S$ is a full assignment to all state variables.

[*] Equal contribution.

[†] This work is partially supported by the National Science Foundation under grants 2113704 and 2148177.

We say s is a ϕ-state if s satisfies a formula ϕ modulo the underlying theory. A formula ψ *implies* another formula ϕ, denoted by $\psi \Rightarrow \phi$, if all state satisfying ψ also satisfies ϕ. A finite or infinite *path* of M is a sequence of states such that the first state is an I-state and all consecutive steps satisfy $T(X, X')$. A path is a ϕ-path if all states along the path are ϕ-states.

M satisfies a safety property P if all paths of M are P-paths. A prominent method to prove safety is through finding an *inductive invariant*, Inv, such that

$$\text{(a) } I \Rightarrow Inv, \text{ (b) } Inv \wedge T \Rightarrow Inv', \text{ (c) } Inv \Rightarrow P. \tag{1}$$

We require M to satisfy a progress property: its output value should eventually change unless it reaches a designated terminal state. This property can either be enforced by the designer or checked by a standard tool [14].

B. Non-cycle-accurate Sequential Equivalence

A sequence π *stutters* at step k if it keeps the same value for indices k and $k+1$. For instance, the sequence $\langle a, a, b, c, \cdots \rangle$ stutters at step 1. We denote the *stutter-free sequence* of π as $\natural\pi$, which is obtained by removing all stuttering steps from π. Two sequences π_1 and π_2 are *equivalent modulo stuttering*, denoted as $\pi_1 \simeq \pi_2$, if $\natural\pi_1 = \natural\pi_2$ [11]. For instance, $\natural\langle a, a, b, c\rangle = \langle a, b, c\rangle$, and $\langle a, a, b, c\rangle \simeq \langle a, b, c, c\rangle$.

A hardware design can be refined across different abstraction levels or optimized with sequential transformation techniques. A pair of designs before and after a *sequential transformation* have the same functionality if they always exhibit the same observable behavior modulo timing differences:

Definition 1. *M_A and M_B are observational equivalent if and only if for every valid input sequence, the corresponding output sequences are equivalent modulo stuttering.*

C. Refinement Relation

The VLSI design flow can be viewed as an iterative refinement process. Each refinement step modifies the design while it preserves the original functionality. A *refinement relation* captures the correspondence between the states of the two designs before and after a sequential transformation:

Definition 2. *$R \subseteq S_A \times S_B$ is a refinement relation for M_A and M_B if the following conditions hold:*

- $I_\times(s_A, s_B) \Rightarrow (s_A \, R \, s_B)$, (2a)
- $(s_A \, R \, s_B) \Rightarrow (s'_A \, R \, s'_B \,\vee\, s'_A \, R \, s_B \,\vee\, s_A \, R \, s'_B)$, (2b)
- $(s_A \, R \, s_B) \Rightarrow P_\times(s_A, s_B)$. (2c)

$I_\times$ represents the *initial correspondence* of the two designs. As discussed in §II-A, the input sequence can be encoded into the initial states of both designs. In this situation, the initial correspondence requires that the X_{in} parts are identical and the X_{state} parts are the corresponding reset states. $I_\times$ can also be specified by the user. $P_\times$ denotes the *observational equivalence property*, defined as $P_\times(s_A, s_B) \triangleq (s_A.out = s_B.out)$, where out represents the output value of the associated state. If some intermediate states are considered non-observable, a special value τ can be assigned to their output values.

The refinement relation in Definition 2 is a bisimulation equivalence relation. It states that *a)* all pairs satisfying the initial correspondence are related by R; *b)* for every pair related by R, there exists a stuttering pattern such that the next pair also satisfies R; *c)* for all pairs related by R, the two designs produce the same output. The existence of a refinement relation implies functional equivalence of the two designs:

Listing 1 $gcd_A(x, y)$

```
while (x − y) ≠ 0 do
    x ← (x−y) > 0 ? (x−y) : x
    y ← (y−x) > 0 ? (y−x) : y
output x
```

Listing 2 $gcd_B(x, y)$

```
while (x − y) ≠ 0 do
    x ← (x−y) > 0 ? (x−y) : y
    y ← (x−y) > 0 ? y : x
output x
```

Lemma 3. *M_A and M_B are observational equivalent if there exists a refinement relation R for M_A and M_B.*

Proof. The existence of a refinement relation ensures that for any input sequence (2a), there is a way to insert stuttering steps (2b), such that the two designs produce identical output sequences (2c) denoted by π. Inserting infinite stuttering steps to either M_A or M_B before they terminate, while maintaining R, would contradict with their progress property. This ensures $\pi_A \simeq \pi$ and $\pi_B \simeq \pi$, and hence $\pi_A \simeq \pi_B$. □

D. Problem Statement

This paper re-investigates the non-cycle-accurate sequential equivalence checking problem. Our objective is to devise an efficient model checking algorithm that can solve both equivalent and non-equivalent instances. Moreover, the algorithm should compute a compact and human-readable refinement relation for equivalent instances.

E. Running Example

The Euclid's algorithm computes the *greatest common divisor (gcd)* of two natural numbers. It iteratively subtracts the smaller number from the greater one until the two numbers are equal. gcd_A (Listing 1) and gcd_B (Listing 2) are two RTL implementations of the algorithm. In gcd_A, two subtractors are initiated simultaneously to compute both $x - y$ and $y - x$. In gcd_B, the designer allocates only one subtractor, so the design needs to switch x and y when x is less than y.

The two designs have the same functionality but are different in timing. Suppose the initial values of x and y are 5 and 1, respectively. Both designs take 4 clock cycles to reach termination. On the other hand, suppose the initial values are 5 and 3. gcd_A takes 3 cycles $\langle(5,3),(2,3),(2,1),(1,1)\rangle$, while gcd_B takes 5 cycles $\langle(5,3),(2,3),(3,2),(1,2),(2,1),(1,1)\rangle$. Notice that the two designs have a *variable period* respect to each other depending on the inputs. Furthermore, there exists no static state mapping or internal node mapping between the two designs.

A refinement relation can capture the correspondence between the states of the two designs, and thus prove their equivalence. For example, $R \triangleq (x_A = x_B \wedge y_A = y_B) \vee (x_A = y_B \wedge x_B = y_A)$ is a simple refinement relation satisfying all the conditions (2a)-(2c).

III. The Refinement Relation Searching Algorithm

A. Overview

This section presents RE3, a rigorous and efficient algorithm to check observational equivalence. Given two hardware designs M_A and M_B, it generates a compact and human-comprehensible refinement relation if they are equivalent, or an input sequence as the counterexample if they are non-equivalent. The general workflow of RE3 is shown in Fig. 1.

RE3 first constructs a *composed transition system* $M_\times$ from M_A and M_B (§III-B). It then leverages a new abstraction paradigm, *relational mapping abstraction*, to implicitly build an abstract model for $M_\times$ (§III-E). On the same abstract domain, the algorithm synthesizes a *scheduling function* that controls the alignment of $M_\times$ (§III-B). Afterward, it launches an IC3-style model checking procedure to find

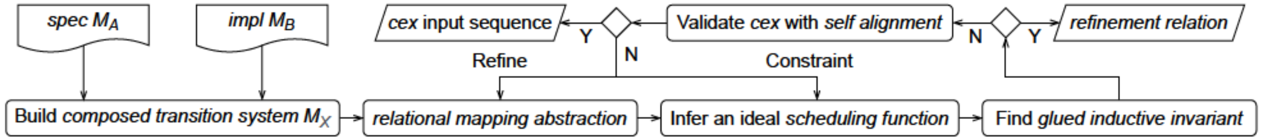

Fig. 1: The general workflow of the RE3 algorithm.

a *glued inductive invariant* $Inv_\times$ (§III-F). If that procedure succeeds, the inductive invariant contains a *relational transition diagram*, which represents the desired refinement relation (§III-D). If that procedure fails, RE3 launches a simulation on the concrete domain using the *self-alignment mechanism* (§III-C) to validate the abstract counterexample. If that validation passes, the algorithm returns an input sequence corresponding to the counterexample. Otherwise, it refines either the scheduling function or the abstract domain to eliminate the spurious counterexample and repeat the verification process.

B. Composed Transition System

Self composition [15], [16] is recently developed to verify hypersafety properties on software programs. We employ self composition to divide the sequential equivalence checking problem into two separate problems: *i)* inferring a *scheduling function* f_s for $M_\times$, and *ii)* finding a glued inductive invariant $Inv_\times$ with respect to f_s. For each *glued state* $s_\times \in S_\times$, where $S_\times \triangleq S_A \times S_B$, f_s controls which of M_A and M_B will move forward in the next clock cycle by selecting a *stuttering pattern*:

Definition 4. *A scheduling function* $f_s : S_\times \to \{M_A, M_B, M_A \cup M_B\}$ *maps each glued state to a non-empty set of designs that are to execute in the next clock cycle.*

It is usually the case that a high-level specification M_A always runs no slower than its implementation M_B, as the former typically has fewer resource constraints. In this situation, we let f_s output a Boolean value: `true` represents a *synchronizing* transition where both M_A and M_B move forward, and `false` represents a *stuttering* transition where only M_B moves forward. Synthesizing f_s thus becomes a binary classification problem in the glued state space.

A composed transition system $M_\times$ is a parallel composition of M_A and M_B. It executes deterministically when augmented with f_s:

Definition 5. *Given two designs* M_A, M_B *and a scheduling function* f_s*, the composed transition system* $M_\times(f_s)$ *is a tuple* $\langle X_A \cup X_B, I_\times, T_\times \rangle$*, where* $I_\times$ *is the initial correspondence (§II-C), and*

$T_\times(s_A, s_B, s'_A, s'_B) \triangleq$
$\wedge$ `if` $M_A \in f_s(s_A, s_B)$ `then` $T_A(s_A, s'_A)$ `else` $s'_A = s_A$
$\wedge$ `if` $M_B \in f_s(s_A, s_B)$ `then` $T_B(s_B, s'_B)$ `else` $s'_B = s_B$.

In Definition 5, the output of f_s at a certain glued state controls whether to stall a branch of $M_\times$. With $M_\times(f_s)$, the problem of finding a refinement relation can be reduced to an ordinary safety checking problem:

Lemma 6. *There exists a refinement relation* R *for* M_A *and* M_B*, if there exists a scheduling function* f_s *and a glued inductive invariant* $Inv_\times$ *for* $M_\times(f_s)$*, such that the following conditions hold:*
- $I_\times \Rightarrow Inv_\times$, *(6a)*
- $Inv_\times \wedge T_\times \Rightarrow Inv'_\times$, *(6b)*
- $Inv_\times \Rightarrow P_\times$. *(6c)*

Proof. Let R equal $Inv_\times$, *i.e.*, for each $s_\times$ within $Inv_\times$, the corresponding s_A and s_B are related by R. Conditions (2a) and (2c) can be immediately derived from (6a) and (6c). Meanwhile, f_s explicitly selects a stuttering pattern for each $s_\times$ to form $T_\times$. Therefore, (2b) can also be derived from (6b). □

Lemma 6 suggests a method to convert the non-cycle-accurate equivalence checking problem to a safety checking problem. In §III-F, we present a comprehensive model checking algorithm to find both f_s and $Inv_\times$ for hardware designs.

C. Self-alignment Mechanism

A glued inductive invariant should not contain any *bad states* that will inevitably reach a $\neg P_\times$-state. With the *self-alignment mechanism*, we can quickly determine whether a concrete glued state $s_\times$ is a bad state. In comparison, property directed self composition (PDSC) [15], [16] needs to account for all stuttering patterns. Specifically, it constructs a tree whose root is $s_\times$ and whose edges are the valid stuttering patterns. Then it traverses the tree in a reverse topological order. A state is labeled as a bad state if it is a $\neg P_\times$-state or if all of its successors are bad states.

The self-alignment mechanism considers a single stuttering pattern for each glued state. Specifically, if the output value of one design changes and that of the other design remains the same, the mechanism stalls the first design while it executes the second design in the next clock cycle. Otherwise, it executes both designs simultaneously in the next clock cycle. Kairos [2] uses a similar strategy to verify hardware modules, but it assumes that handshake signals are always available.

Theorem 7 describes the relationships among observational equivalence, scheduling function, glued inductive invariant, and the self-alignment mechanism. We omit the proof due to space limit.

Theorem 7. *The following statements are always equivalent:*
- M_A *and* M_B *are observational equivalent;* *(7a)*
- *A refinement relation* R *exists for* M_A *and* M_B*;* *(7b)*
- *There exists a scheduling function* f_s *and a corresponding glued inductive invariant* $Inv_\times$ *for* $M_\times$*;* *(7c)*
- *There exists a scheduling function* f_s *such that all paths of* $M_\times(f_s)$ *are observational equivalent paths;* *(7d)*
- *All paths produced by the self-alignment mechanism on* $M_\times$ *are observational equivalent paths.* *(7e)*

Lemma 8. *A glued state* $s_\times$ *is a bad state if and only if the two output sequences produced by the self-alignment mechanism starting from* $s_\times$ *are different.*

Lemma 8 can be proved by treating $s_\times$ as $I_\times$ and using the same reasoning as the *(7a)* $\Rightarrow$ *(7e)* proof in Theorem 7. It suggests that we can check whether $s_\times$ is bad by running a simulation with the self-alignment mechanism from $s_\times$.

Based on Theorem 7, a viable approach to verify observational equivalence is to launch model checking on the composed transition system augmented by the self-alignment mechanism. However, the main challenge lies in the "quality" of the scheduling function. Some scheduling functions admit easier-to-find inductive invariants, which can lead to more concise refinement relations. RE3 simultaneously synthesizes the scheduling function and finds the corresponding inductive invariant to accelerate convergence and maximize the chance of finding a simple inductive invariant.

D. Relational Transition Diagram

Abstraction techniques can reduce the complexity of the model checking problem while preserving the facts relevant to the problem. Consider an abstraction function $\alpha : S_\times \to \widehat{S_\times}$ that maps concrete glued states into abstract ones. There is an abstract transition between two abstract states $\widehat{s}_1$ and $\widehat{s}_2$ if there is a concrete transition between any pair of concrete states s_1 and s_2, such that $\alpha(s_1) = \widehat{s}_1$ and $\alpha(s_2) = \widehat{s}_2$. Suppose we have found a glued inductive invariant $\widehat{Inv_\times}$ on an abstract domain. Then we can construct a *relational transition diagram* as follows:

Definition 9. *A relational transition diagram is a directed graph whose vertices are all the abstract glued states within $\widehat{Inv_\times}$ and whose edges are the abstract transitions between these states.*

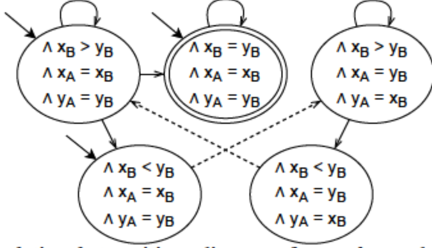

Fig. 2: A relational transition diagram for gcd_A and gcd_B when $f_s := x_B \geq y_B$? **true** : **false**. Solid (resp. dashed) arrows represent *synchronizing* (resp. *stuttering*) transitions. An input arrow indicates an initial state and a double circle indicates a terminal state.

As an illustrative example, Fig. 2 shows a relational transition diagram for gcd_A and gcd_B. Recall that gcd_A is always no slower than gcd_B. We choose a scheduling function that executes both designs when $x_B \geq y_B$ and executes only gcd_B otherwise. This scheduling function leads to a simple diagram that consists of only five abstract states.

Because a relational transition diagram is constructed from a glued inductive invariant and a valid scheduling function, it induces a transition relation R (Lemma 6). Two states $s_A \in S_A$ and $s_B \in S_B$ are related by R if they are glued together, *i.e.*, $\alpha((s_A, s_B))$ is in the diagram. In the gcd example, $s_A = (5, 3)$ and $s_B = (5, 3)$ are glued together by the top-left state in Fig. 2.

In the rest of this section, we will discuss how to construct an abstract domain that allows a compact relational transition diagram, and how to efficiently search for a glued inductive invariant.

E. Relational Mapping Abstraction

Predicate abstraction [17] and syntax-guided abstraction [13] are two prevalent abstraction schemes for hardware verification. Predicate abstraction introduces a set of predicates over the state variables. A predicate-abstracted state represents all those concrete states with the same valuation to the predicates. Syntax-guided abstraction utilizes the terms in the syntax of the design to encode the state space. A syntax-abstracted state is a *partition assignment* that captures all equality and inequality among the terms. For the gcd_A example, the concrete state $(2, 4)$ can be mapped to the abstract state $(\neg(x-y) > 0) \wedge ((y-x) > 0) \wedge \{x, y-x \mid y \mid x-y\}$, where vertical bars divide terms and formulas into equivalent classes.

During our experiments on hardware equivalence checking, we make several key observations on abstraction schemes. First, M_A and M_B operate on two disjoint sets of state variables. Therefore, we introduce two separate sets of predicates over the corresponding sets of state variables. This can simplify the generation of new predicates and thus reduce the search space of a model checking algorithm. Second, we notice that latches* are the first-class citizens in an equivalence checking problem. A conditional latch mapping almost always exists for a pair of equivalent designs [9]. Hence, an abstraction scheme can benefit from explicitly capturing the one-to-one correspondence between a subset of latches from both designs. Third, we notice that the equivalence of latches is more important than the non-equivalence of latches. Thus, we only add constraints for the pairs of equivalent of latches. Compared with syntax-guided abstraction, this approach can usually significantly reduce the number of abstract states.

In light of these observations, we propose a novel abstraction scheme, *relational mapping abstraction* (RMA), that is dedicated to hardware equivalence checking. An RMA abstracted state is represented by an *r-cube* consisting of three fields: *r-cube* $\triangleq pred_A \wedge pred_B \wedge mapping$. Each field is either empty or a conjunction of atomic predicates. The state variables of M_A can appear only in $pred_A$ and the left-hand side of $mapping$, whereas the state variables of M_B can appear only in $pred_B$ and the right-hand side of $mapping$. Intuitively, the $mapping$ field explicitly expresses the latch mapping, while $pred_A$ and $pred_B$ specify the conditions for that mapping.

A glued inductive invariant $\widehat{Inv_\times}$ on an RMA abstract domain is represented as a disjunction of r-cubes. Such an inductive invariant naturally gives rise to a relational transition diagram (§III-D). For instance, every node in Fig. 2 is represented by an r-cube. The disjunction of all those r-cubes comprises an inductive invariant of the the composed transition system $M_\times(f_s)$.

F. Finding Refinement Relation with RE3

Algorithm 1 The RE3 algorithm.

- **Input:** high-level design M_A, low-level design M_B, initial correspondence $I_\times$.
- **Output:** a relational transition diagram $D_\times$ that represents the refinement relation if M_A and M_B are observational equivalent; a concrete witness of non-equivalence $s^*_\times \in I_\times$ otherwise.

```
1: Construct M× from MA and MB                              ▷ §III-B
2: Initialize α with initial predicates        ▷ abstract function, §III-E
3: while true do
4:   Fs ← ⊤        ▷ reset constraints on scheduling function, §III-B
5:   while true do
6:     (reachable, Inv×, cex) ← isReachable(M×(Fs), α)
7:     if reachable = false then          ▷ found inductive invariant
8:       D× ← buildDiagram(Inv×, Fs, α)                     ▷ §III-D
9:       return D×
10:    else                          ▷ found abstract counterexample trace
11:      (spurious, s*×) ← isSpurious(cex)
12:      if spurious = false then return s*×                ▷ witness
13:      else
14:        Fs ← addConstraints(Fs, cex, α)
15:        if Fs is invalid then break
16:  α ← refine(α)
```

The RE3 Algorithm. A high-level description of RE3 is given in Algorithm 1. Leveraging the implicit abstraction technique [17], RE3 exploits relational mapping abstraction throughout its execution, and we omit *hat* symbols in the pseudocode to improve readability. The algorithm starts by constructing a composed transition system $M_\times$ from the two designs (Line 1). If the user has extracted a set of control statements from the designs, RE3 can initialize the abstract domain

*In this context, "latches" can refer to any memory elements at either bit-level or word-level.

with those statements (Line 2). The algorithm initiates a nested loop to find a compact refinement relation. In every iteration of the outer loop, RE3 incrementally refines a non-deterministic set of scheduling functions F_s, which is initially unconstrained. In every iteration of the inner loop, RE3 calls an IC3-style [18], [19] model checking procedure on $M_\times$ augmented by F_s (Line 6). If $\neg P_\times$ is unreachable from $I_\times$ on the current abstract domain, a glued inductive invariant $Inv_\times$ is returned by the model checking procedure. In this case, RE3 builds a relational transition diagram (Line 8) that represents the refinement relation and provides it to the user. Otherwise, an abstract counterexample trace is returned by the model checking procedure. RE3 validates the counterexample with the self-alignment mechanism (Line 11). If the validation passes, a concrete glued state $s^*_\times \in I_\times$ is extracted as a witness to prove that M_A and M_B are not observational equivalent (Line 12). If the counterexample turns out to be spurious, RE3 strengthens the constraints on F_s to eliminate the counterexample and continues to the next iteration (Line 14). However, if F_s is no longer *valid* after adding the new constraints (Line 15), RE3 exits to the outer loop and refines the abstract domain (Line 16). It resets the constraints on F_s before it enters the inner loop again (Line 4).

In the following, we highlight several key components and features of the RE3 algorithm.

• RE3 reverses the sequence of frames so that the relational mapping abstraction scheme can be compatible with the IC3 framework. Recall that the RMA scheme represents a glued inductive invariant as a disjunction of r-cubes. On the other hand, a frame in the IC3 framework is a conjunction of clauses. RE3 takes the negation of the former during model checking to ensure compatibility. Accordingly, RE3 reverses the order of the frames, such that $F_0 = \neg P_\times$ and $F_i \Rightarrow \neg I_\times$.

• RE3 navigates the searching process towards finding a concise refinement relation. It always starts from an unconstrained set of scheduling functions and gradually refines the set with spurious counterexamples. By controlling the granularity, the algorithm is more likely to find a general and compact inductive invariant. Besides, RE3 prioritizes the refinement of scheduling functions over the refinement of the abstract domain. The latter is executed only when necessary, *i.e.*, all the remaining scheduling functions are invalid. We adopt this strategy because the latter dominates the complexity of model checking and the refinement relation, while the former can be restored periodically.

• RE3 utilizes a lightweight simulation-based approach to find bad states and thus accelerate convergence. Whenever a counterexample trace is returned by the *isReachable* procedure, or a proof obligation is added to the priority queue, RE3 randomly samples N concrete glued states within the targeted abstract state. RE3 runs a simulation on the concrete domain with the self-alignment mechanism (§III-C) from each of those states. All resulting bad states are permanently recorded and utilized by all subsequent invocations of the *isReachable* procedure. Because this approach is speculative, missing a bad state during sampling does not affect the soundness of RE3.

• RE3 ensures that the model checking procedures in every iteration of the outer loop are fully incremental. Each time an abstract counterexample trace is returned by the *isReachable* procedure, RE3 analyses the trace and adds new scheduling constraints to F_s. During this process, the transition relation of $M_\times(F_s)$, denoted as $T_\times$, is monotonically strengthened. This ensures that the whole sequence of frames can be reused in the next execution. Furthermore, RE3 labels a glued abstract state as a *temporary bad state* if every one of its successors is a temporary bad state or contains a concrete bad state. Once an $I_\times$-state is contained in a temporary bad state, F_s becomes *invalid*, *i.e.*, it is impossible to find an $Inv_\times$ for any of the remaining scheduling functions within F_s. In that case, RE3 resets F_s, removes all temporary bad states, and calls the *refine* procedure.

• RE3 employs an efficient strategy to eliminate the counterexample tree and thus refine the abstract domain. Briefly speaking, it walks along a single path from the root until it finds a suitable abstract state to split. For $pred_A$ and $pred_B$, it leverages *interpolation* to generate new predicates to split the targeted state. For the $mapping$ field, it collects a set of potentially equivalent latch pairs by simulating the two designs, utilizes *unsat core* extraction to find a minimal set of those pairs, and adds them to $mapping$. Finally, RE3 chooses the simplest one among the three fields to settle down the refinement.

Correctness and Termination. We rely on the correctness of the original IC3 algorithm to establish the correctness of the RE3 algorithm. When RE3 returns a refinement relation, it must have found a valid scheduling function and an associated inductive invariant. According to Theorem 7, M_A and M_B must be observational equivalent. On the other hand, the two designs must be non-equivalent when RE3 returns a concrete witness of non-equivalence that satisfies the initial correspondence. This is guaranteed by the correctness of the self-alignment mechanism.

No complexity bounds exist for model checking algorithms if the composed transition system is in the infinite state space. Nevertheless, if all variables of the two designs are constant-sized bit vectors, RE3 will eventually terminate. This is because the abstraction refinement process is strictly monotonic, while the scheduling function pruning process and the IC3 converging process are also strictly monotonic on a fixed abstract domain.

IV. Evaluation

A. Experimental Results

We implemented RE3 in Python on top of the Boolector SMT solver [20]. We selected 7 high-level hardware specifications from the HLSynth benchmark suite [21]. The same set of specifications is used in the SE3 paper [3]. We leveraged a high-level synthesis tool, Xilinx Vivado HLS, to generate 6 RTL designs with different timing for each specification. Thus, we obtained 105 pairs of equivalent designs for all those specifications. Moreover, we manually introduced errors into the designs to create 105 pairs of non-equivalent designs. All experiments are conducted on a Linux machine with a 3.2GHz processor. Each instance is executed on a single thread with a 4GiB of memory limit and a 7,200-second timeout.

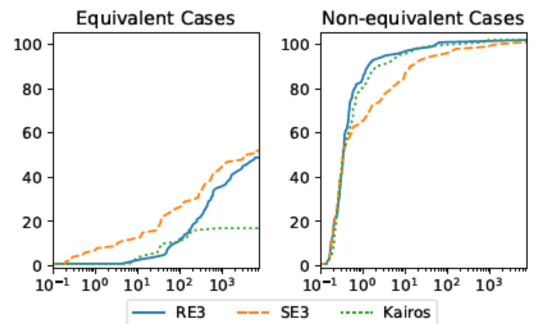

Fig. 3: Comparing the number of solved instances over time *(s)*.

Figure 3 compares RE3 against two existing non-cycle-accurate sequential equivalence checking algorithms, SE3 and Kairos [2], for their efficiency. On the equivalent benchmarks, RE3 and SE3 are

significantly faster than Kairos. RE3 is slightly slower than SE3 on these benchmarks. We believe this is because SE3 adopts a monolithic strategy that solves the scheduling function, the inductive invariant, and the reachability information at the same time. In comparison, RE3 discards some of the previous results when it starts a new iteration of the outer loop. However, such an overhead is justified by the fact that RE3 can find a refinement relation in addition to the verification result. On the non-equivalent benchmarks, both RE3 and Kairos are faster than SE3. It is because RE3 employs the simulation-based self-alignment mechanism, and Kairos always concentrates on a fixed alignment pattern throughout the model checking process.

We measured the size of an inductive invariant by counting the number of cubes (§II-A) within its logic formula. We collected all pairs of equivalent designs that can be solved by both RE3 and SE3. In Figure 4, each cross mark represents such a pair of designs. As plotted in the figure, the inductive invariants produced by RE3 are almost always more concise than those produced by SE3. The average size of the former is 10.3 cubes, whereas the average size of the latter is 33.1 cubes. In this regard, RE3 achieves a 69% reduction in size on average.

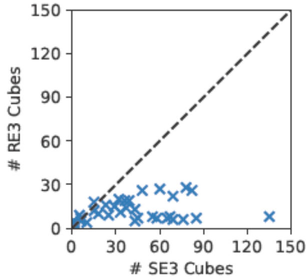

Fig. 4: Comparing the size of inductive invariants: RE3 vs. SE3.

B. Case Study

The following study showcases how a concise and explainable refinement relation can benefit IC design and verification. Recently, an abundant literature focuses on leveraging large language models (LLMs) for RTL code generation [22]–[25]. They rely on user feedback and prompt engineering to iteratively refine a design and improve its quality. Nevertheless, LLMs have an Achilles heel: their tendency to *hallucinate*. They can generate designs that seem plausible but contain subtle functional errors [24]. It is notoriously difficult to detect and fix such errors.

Existing work relies on simulation to determine the correctness of the generated design [22]–[25]. However, an error trace produced by a simulator does not reveal the root cause of the error. According to our experiments with *GPT-4o* [26], an LLM can be trapped in repeated interactions with a simulator because it is incapable of interpreting and reasoning about the error traces.

We propose a comprehensive method to detect and fix errors in LLM-generated designs. Starting from a reference design D_0, it incrementally applies a sequence of transformations with LLM to optimize the performance and incorporate new features. Suppose it has verified the correctness of D_i, the i-th version of the design, and obtained R_i, the refinement relation between D_0 and D_i. Given a new version D_{i+1}, our method traverses the relational transition diagram associated with R_i to check whether D_{i+1} violates any of the transitions in the diagram. Upon detecting a violation, the transition, its start state, and its end state are encapsulated in a prompt for the LLM to fix that error. When D_{i+1} is proved correct, the method computes a new refinement relation R_{i+1} and proceeds to the next iteration.

$x \leftarrow x \ll 1$
$y \leftarrow y \gg 1$
$z \leftarrow y[0]$ **?** $z + x$ **:** z

(a) $multi_A$

$x \leftarrow y[1]$ **?** $x \ll 1$ **:** $x \ll 2$
$y \leftarrow y[1]$ **?** $y \gg 1$ **:** $y \gg 2$
$z \leftarrow y[0]$ **?** $z + x$ **:** z

(b) $multi_B$

if $y[3{:}1] = 000$ **then**
  $x \leftarrow x \ll 4;\ y \leftarrow y \gg 4$
else if $y[2{:}1] = 00$ **then**
  $x \leftarrow x \ll 3;\ y \leftarrow y \gg 3$
else if $y[1] = 0$ **then**
  $x \leftarrow x \ll 2;\ y \leftarrow y \gg 2$
else $x \leftarrow x \ll 1;\ y \leftarrow y \gg 1$
$z \leftarrow y[0]$ **?** $z + x$ **:** z

(c) $multi_C$ (expected)

if $y[3{:}0] = 0001$ **then**
  $x \leftarrow x \ll 1;\ y \leftarrow y \gg 1$
else if $y[3{:}0] = 0010$ **then**
  $x \leftarrow x \ll 2;\ y \leftarrow y \gg 2$
else if $y[3{:}0] = 0100$ **then**
  $x \leftarrow x \ll 3;\ y \leftarrow y \gg 3$
else if $y[3{:}0] = 1000$ **then**
  $x \leftarrow x \ll 4;\ y \leftarrow y \gg 4$
else $x \leftarrow x \ll 1;\ y \leftarrow y \gg 1$
$z \leftarrow y[0]$ **?** $z + x$ **:** z

(d) $multi_D$ (incorrect)

Fig. 5: Loop bodies of 4 implementations of `multi_16bit` benchmark.

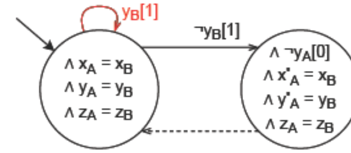

Fig. 6: A relational transition diagram for $multi_A$ and $multi_B$. x'_A and y'_A denote the next states of x_A and y_A, respectively.

We use `multi_16bit` from the *RTLLM* benchmark suite [23] to illustrate the above-mentioned method. Fig. 5(a) displays the simplified loop body of the reference design given by *RTLLM*. In every clock cycle, the multiplicand x is shifted to the left by 1 bit. If the last bit of the multiplier y is 1, the product z accumulates the shifted value of x. In practice, one can use a high-level specification, a certified initial implementation, or a transactional-level model (TLM) as the reference design. Suppose the designer decides to improve the performance of the design by looking into the next 2 bits instead of 1. Specifically, both x and y will be shifted twice if $y[1]$ is 0. Fig. 5(b) displays the pseudocode of the loop body generated by *GPT-4o*. Notice that this transformation is non-cycle-accurate since the timing of $multi_B$ is dependent on input values. We run RE3 on $multi_A$ and $multi_B$, yielding a concise refinement relation R_B. The corresponding relational transition diagram is shown in Fig. 6.

Suppose the designer wants to improve the performance further by looking into the next 4 bits. The two operands will be shifted multiple times if there is a consecutive sequence of zeros in y. Fig. 5(c) displays a correct implementation. Nevertheless, the implementation generated by *GPT-4o* (Fig. 5(d)) is incorrect and cannot pass the testbench provided by *RTLLM*. We attempted to prompt the LLM with the error traces, but it could not identify the root cause from this information. Hence, we traverse the relational transition diagram of R_B to check whether $multi_D$ violates any of those transitions. As shown in Fig. 6, the transition marked in *red* is violated. In a nutshell, $multi_D$ overlooked that the shifting behavior should not depend on the value of $y[0]$. We prompted *GPT-4o* with the transition and the associated states, which enabled the LLM to detect and fix the error.

V. Conclusion

In this paper, we present a new method for non-cycle-accurate sequential equivalence checking. This method can automatically synthesize a refinement relation during its execution. We illustrate potential applications of the refinement relation in the IC design and verification flow. We plan to explore additional use cases and integrate the refinement relation into EDA tools in the future.

References

[1] S. Dai, G. Liu, R. Zhao, and Z. Zhang, “Enabling adaptive loop pipelining in high-level synthesis,” in *ACSSC 2017*, pp. 131–135.

[2] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, “Kairos: Incremental verification in high-level synthesis through latency-insensitive design,” in *FMCAD 2019*.

[3] Y. Li, G. Zhao, Y. He, and H. Zhou, “Se3: Sequential equivalence checking for non-cycle-accurate design transformations,” in *DAC 2023*.

[4] B. Godlin and O. Strichman, “Regression verification,” in *DAC 2009*.

[5] R. Dureja and K. Y. Rozier, “Fuseic3: An algorithm for checking large design spaces,” in *FMCAD 2017*.

[6] J. Shi and S. A. Seshia, “Lifting hardware models from implementations for verification,” Master’s thesis, EECS Department, University of California, Berkeley, 2022.

[7] P. Subramanyan, B.-Y. Huang, Y. Vizel, A. Gupta, and S. Malik, “Template-based parameterized synthesis of uniform instruction-level abstractions for soc verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, pp. 1692–1705, 2017.

[8] A. Bernardini, W. Ecker, and U. Schlichtmann, “Where formal verification can help in functional safety analysis,” in *ICCAD 2016*.

[9] W. Fang, G. Hu, and H. Zhang, “r-map: Relating implementation and specification in hardware refinement checking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[10] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, “Ilang: A modeling and verification platform for socs using instruction-level abstractions,” in *TACAS 2019*.

[11] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[12] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Parameter synthesis with ic3,” in *FMCAD 2013*.

[13] A. Goel and K. Sakallah, “Model checking of verilog rtl using ic3 with syntax-guided abstraction,” in *NASA Formal Methods 2019*.

[14] J. Daniel, A. Cimatti, A. Griggio, S. Tonetta, and S. Mover, “Infinite-state liveness-to-safety via implicit abstraction and well-founded relations,” in *CAV 2016*.

[15] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, “Property directed self composition,” in *CAV 2019*.

[16] A. Shenoy, S. Prabhu, K. Madhukar, R. Shemer, and M. Srivas, “Automated property directed self composition,” in *ATVA 2023*.

[17] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Ic3 modulo theories via implicit predicate abstraction,” in *TACAS 2014*.

[18] A. R. Bradley, “Sat-based model checking without unrolling,” in *VMCAI 2011*.

[19] N. Een, A. Mishchenko, and R. Brayton, “Efficient implementation of property directed reachability,” in *FMCAD 2011*.

[20] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, “Btor2, btormc and boolector 3.0,” in *ICCAD 2018*. Springer, pp. 587–595.

[21] P. R. Panda and N. D. Dutt, “1995 high level synthesis design repository,” in *ISSS 1995*, pp. 170–174.

[22] M. Liu, N. Pinckney, B. Khailany, and H. Ren, “Verilogeval: Evaluating large language models for verilog code generation,” in *ICCAD 2023*.

[23] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, “Rtllm: An open-source benchmark for design rtl generation with large language model,” in *ASP-DAC 2024*, pp. 722–727.

[24] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, “Meic: Re-thinking rtl debug automation using llms,” *ICCAD 2024*.

[25] X. Wang, G.-W. Wan, S.-Z. Wong, L. Zhang, T. Liu, Q. Tian, and J. Ye, “Chatcpu: An agile cpu design and verification platform with llm,” in *DAC 2024*.

[26] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.