SE3: Sequential Equivalence Checking for Non-Cycle-Accurate Design Transformations[†]

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*—In high-level design explorations, many useful optimizations transform a circuit into another with different operating cycles for a better trade-off between performance and resource usage. How to efficiently check their equivalence is critical and challenging since most existing equivalence checkers are designed for cycle-accurate circuits. This paper presents SE3, an efficient sequential equivalence checker without assumption on cycle-accuracy, latch mapping, or I/O interface of the checked circuits. It proves the equivalence of two circuits by computing an equivalence relation between the states of the two circuits and utilizes syntax abstraction to accelerate this process. Experimental results show that SE3 is significantly faster than state-of-the-art sequential equivalence checking algorithms.**

***Index Terms*—equivalence checking, model checking, sequential circuit, quantifier elimination**

I. Introduction

With the growing demand for high-performance integrated circuits, design engineers and optimization tools tend to perform more aggressive sequential transformations to meet the timing and throughput goals. For instance, retiming techniques move logic across flip-flops to meet timing constraints; pipelining techniques introduce additional pipeline stages, trading off latency for throughput; pre-computation techniques prepare the results earlier to remove their computations from the critical path. Other examples of sequential transformations include unrolling, resource reallocation, clock gating, and memory partitioning. A design after such transformations may no longer be cycle-accurate with the original one. Additionally, there may not exist a one-to-one latch mapping between the two circuits because these transformations can change the functionalities of the flip-flops.

Recently, adaptive pipelining [1], [2] is proposed to enable dynamic scheduling, *i.e.*, the period of each iteration is a variable depending on the inputs and the previous executions. Moreover, the latency-intensive (LI) design methodology has emerged to tolerate the arbitrary timing of individual hardware modules. Under the relaxed timing requirements, engineers can design highly customized hardware modules with *variable latencies*. Sequential equivalence checking is the key enabler of sequential design transformations. It finds application in various stages of the IC design flow [3], for example, in checking if an RTL model conforms to the functional specification it seeks to implement or determining if a derivative RTL or gate-level model is functionally equivalent to a validated model. Co-simulation is also widely used in such scenarios but has limited functional coverage. In comparison, formal sequential equivalence checking is a comprehensive and rigorous approach that allows verification engineers to prove the consistency of two designs over any number of cycles.

Several formal sequential equivalence checking algorithms have been proposed [4]–[8], yet nearly all of them have restricted use cases. Some algorithms assume that a complete latch mapping is provided by the user [4]. Others assume that the two designs are structurally similar and that pairs of internal nodes exist that have identical functionalities [5]–[7]. Kairos [8] can handle almost all types of transformations, but it requires both designs under verification to follow a $valid\text{-}ready$ interface protocol.

In this paper, we propose SE3 (**S**yntax-**E**ncoded **S**tuttering **E**quivalence Checking for **SE**quential Circuits), a general and efficient algorithm based on symbolic model checking. SE3 formulates the sequential equivalence checking problem for any transformations [9] as checking whether the output sequences of the two designs under verification are alignable given a set of corresponding initial states and any input sequences. Nevertheless, the consequent formula involves an alternation of universal and existential quantifiers. Eliminating the internal existential quantifier can result in a formula that is exponential in size.

SE3 tackles this issue by searching for a reversed inductive invariant on a product machine of two designs, bypassing the need for quantifier elimination. Additionally, SE3 maintains a frame structure similar to IC3 [10]. This allows the algorithm to learn new clauses incrementally and enables high flexibility in expressing complex equivalence relations. Moreover, SE3 leverages syntax abstraction [11] to capture the equivalence relations among the internal nodes. Thus, the algorithm concentrates on high-level, coarse-grained relations at the beginning of the verification process and gradually shifts to finer-grained relations through iterative refinement.

We demonstrate the capability and efficiency of SE3 with a case study and a benchmark suite. SE3 is significantly faster than Kairos on equivalent test cases and scales well regarding word size. Besides, SE3 is capable of discovering concise and essential inductive invariants, which may guide design engineers to understand the nature of the transformations or locate errors in future transformations.

Our main contributions are:
- We devise a formal property to determine whether two designs are observational equivalent modulo stuttering;
- We utilize the reversed inductive invariant to bypass the quantifier alternation issue within the property;
- We adapt the IC3 symbolic model checking framework to achieve incremental verification;
- We embed syntax abstraction to the stuttering equivalence checking algorithm, so that SE3 can discover equivalence relations among state variables and internal nodes at various granularities;
- We evaluate SE3 against state-of-the-art sequential equivalence checking algorithms.

II. Background

A. A Running Example

We use the Euclidean algorithm as the running example throughout this paper. To compute the greatest common divisor of two integers, the algorithm iteratively subtracts the smaller integer from the greater one until they become equal. Listing 1 shows an RTL implementation of the algorithm. Two subtractors are initiated in parallel, and only one integer is updated depending on the results. Suppose an

*Equal contribution.
†This work is partially supported by the National Science Foundation under grants 2113704 and 2148177.

Listing 1 Euclidean $gcd_A(x, y)$

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

Listing 2 Euclidean $gcd_B(x, y)$

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

engineer then decides to allocate only one subtractor and adjusts the implementation accordingly. As shown in Listing 2, only one subtractor is initiated in a clock cycle.

The two implementations are identical in function but different in timing. If x equals 6 and y equals 2 initially, both circuits will take 2 cycles to reach convergence. On the other hand, if the initial values of x and y are 6 and 10, gcd_A will take 3 cycles $\langle(6, 10),(6, 4),(2, 4),(2, 2)\rangle$, while gcd_B will take 6 cycles $\langle(6, 10),(10, 6),(4, 6),(6, 4),(2, 4),(4, 2),(2, 2)\rangle$. It can be seen that gcd_B has a variable latency relative to gcd_A, and there exists no latch mapping or equivalent internal nodes between the two designs.

B. Preliminaries

We consider standard first-order logic. A *term* is a variable or a *function* symbol. A *predicate* is an expression applied to a tuple of terms and evaluates to a Boolean value. An *atom* is a Boolean variable or a *predicate* symbol. The terms with non-Boolean values are also referred to as *words*. A *formula* is built over atoms with propositional logic. A *literal* is an atom or its negation. A *cube* is a conjunction of literals, while a *clause* is a disjunction of literals.

A transition system M is defined as a tuple $\langle X, I, T\rangle$, where X is a set of *state variables*, X' is 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 variables X'_{in} are unconstrained or controlled by an external specification. A state s is a full assignment to all state variables. We write $s \models \phi$ if s satisfies a formula ϕ modulo the underlying theory, and s is a ϕ-state. A formula ψ *implies* another formula ϕ, $\psi \Rightarrow \phi$, if all state satisfying ψ also satisfies ϕ. A finite or infinite *path* is a state sequence 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. A reachable *lasso-shaped path* [14] (a *lasso*, in shorthand) is a finite run from an initial state followed by a loop.

A sequence σ *stutters* at step k if it keeps the same value for indices k and $k+1$. For instance, the sequence $\langle a, b, b, c, \cdots\rangle$ stutters at step 2. We define $\natural\sigma$ the *stutter-free sequence* of σ, which eliminates all stuttering steps in σ. We let $\sigma \simeq \rho$ mean $\natural\sigma = \natural\rho$ [12]. For instance, $\natural\langle a, b, b, c\rangle = \langle a, b, c\rangle$, and $\langle a, b, b, c\rangle \simeq \langle a, b, c, c\rangle$.

C. The IC3 Model Checking Algorithm

IC3 [10] is the state-of-the-art symbolic model checking algorithm for hardware verification. Given a safety property $P(X)$, IC3 checks whether $M \models P$, *i.e.*, all paths of M are P-paths. In this regard, it tries to find an *inductive invariant*, Inv, such that

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

Once an Inv is found, IC3 completes the proof.

During its execution, IC3 maintains a sequence of *frames* $F_0(X),\ \cdots, F_k(X)$ such that

$$\forall i < k:\ (a)\ F_0 = I,\ (b)\ F_i \wedge T \Rightarrow F'_{i+1},\ (c)\ F_i \Rightarrow P. \tag{2}$$

Additionally, the algorithm ensures that each frame is a conjunction of clauses, and $clauses(F_{i+1}) \subseteq clauses(F_i)$, where $clauses(F_i)$ denotes the set of all clauses that constitute F_i. The algorithm attempts to learn new clauses from the reachability information of the system and add them to the sequence of frames until one of the frames is proved to be an inductive invariant.

In the following, we give a high-level description of the algorithm. At the beginning of every iteration, k is incremented by 1, and a new frame $F_k = P$ is attached to the sequence. The algorithm queries

$$SAT?(F_k \wedge T \wedge \neg P') \tag{3}$$

for a new bad state $s \in F_k$. If (3) is satisfiable, a new *proof obligation* $\langle k, s\rangle$ is added to a priority queue. Whenever the priority queue is non-empty, IC3 pops the top element of the queue and queries

$$SAT?(F_i \wedge T \wedge s'). \tag{4}$$

If (4) is satisfiable, there must exist another bad state $t \in F_i$, which is a predecessor of s. A new obligation $\langle k - 1, t\rangle$ is added to the queue. If (4) is unsatisfiable, s can be safely excluded from F_i without affecting the conditions of the frames. Thus, a new clause $c = \neg s$ is conjoined to F_{i+1}. The unsatisfiability of (4) guarantees that c is *relatively inductive* to F_i, *i.e.*, $F_i \wedge c \wedge T \Rightarrow c'$ is a tautology.

If (3) is unsatisfiable, all states in F_k must be more than 1 step away from bad states. Before IC3 starts a new iteration, it pushes every $c \in clauses(F_i)$ to $clauses(F_{i+1})$ if c is relatively inductive to F_{i+1}. During this process, if two consecutive frames F_i and F_{i+1} become identical, they must satisfy all the conditions of an inductive invariant, *i.e.*, $Inv = F_i$, and that finishes the algorithm.

D. Syntax Abstraction

Syntax abstraction [11] creates an abstract space using a subset of the terms present in the original model. An abstract state is a *partition assignment* which captures Boolean values of atoms and equality relations among the words of each sort. For example, in Listing 1, the concrete state $(6, 3)$ may correspond to the abstract state $((x - y) > 0)\ \wedge\ \neg((y - x) > 0)\ \wedge\ \{x \mid y, x - y \mid y - x\}$, where vertical bars divide terms into equivalence classes.

Syntax abstraction removes irrelevant bit-level details, thus facilitating the reasoning of equivalence relations at a coarse granularity. An abstract space it creates can be iteratively refined in a *counterexample-guided abstraction refinement* (CEGAR) fashion: once a spurious counterexample is found, new terms are introduced to eliminate it. Hence, syntax abstraction can be closely integrated with a model checking algorithm, where the former provides the domain for reasoning and the latter provides the guidance for refinement.

III. Problem Definition and Analysis

A. Problem Definition

Our objective is to check the observational equivalence of two systems, *i.e.*, whether their externally observable behaviors are always identical. More specifically, we aim to devise an efficient symbolic model checking algorithm for the following property: starting from any pair of corresponding initial states, the stutter-free output sequences produced by the two systems are identical given the same input sequence.

B. Product Machine for Stuttering

As the first step to solving the problem, we devise an automated reasoning mechanism that checks whether two output sequences are equivalent modulo stuttering. It is based on the observation that inserting stuttering steps to the faster sequence is the *dual* of

eliminating stuttering steps from the slower sequence. Hence, we build a product machine for stuttering, $M_\times$, to mimic the process of inserting stuttering steps. The product machine converts the problem of checking alignability [9] to the problem of finding a feasible auxiliary input sequence. Denote the two systems under verification as M_A and M_B, where M_A runs no slower than M_B. The state space $S_\times$ of $M_\times$ is a Cartesian product $S_A \times S_B$, and every state $s \in S_\times$ is a pair (u, v) where $u \in S_A$ and $v \in S_B$. The transition relation $T_\times$ is composed of two branches, T_{syn} and T_{stu}, both of which are also product machines. T_{syn} specifies the behavior that M_A and M_B move *syn*chronously: $T_{syn}(u, v, u', v') \triangleq T_A(u, u') \wedge T_B(v, v')$, while T_{stu} specifies the behavior that M_A *stu*tters and M_B moves forward: $T_{stu}(u, v, u', v') \triangleq (u = u') \wedge T_B(v, v')$. $T_\times$ uses a dummy input, sel, to select its next state from the two branches[1]:

$$\begin{aligned} T_\times(u, v, sel, u', v') \triangleq &\vee ((sel = 0) \wedge T_{syn}(u, v, u', v')) \\ &\vee ((sel = 1) \wedge T_{stu}(u, v, u', v')). \end{aligned} \tag{5}$$

We define the *observational equivalence property* to be $P_\times(u, v) \triangleq valid \Rightarrow (u_{out} = v_{out})$. The $valid$ signal is only necessary when the output registers can turn into an unstable observable state. We use $(i_a, i_b) \in I_\times$ to denote that two initial states from both systems are related by the *initial correspondence* $I_\times$. By default, $I_\times$ contains the pair of reset states. For some applications, it might be more *convenient* to directly initialize the corresponding pairs of state variables with the same input values. In our running example, this means setting $I_\times$ to the formula $(x_A = x_B) \wedge (y_A = y_B)$. Advanced users can also write customized specifications for $I_\times$ and $P_\times$.

From a specific initial state, the state sequence produced by M_A or M_B is either a finite or an infinite path. We convert all finite paths to infinite ones by adding a *self-loop* to the final states. Notice that final states can usually be distinguished symbolically with termination conditions, $valid$ signals, or as the deadlock states in deadlock-free systems.

Facilitated by $M_\times$, we can check if two systems are *observational equivalent modulo stuttering* by checking the following condition:

Definition 1. *Two systems M_A and M_B are observational equivalent modulo stuttering if and only if there exists a $P_\times$-lasso on $M_\times$ from every pair $(u_1, v_1) \in I_\times$.*

Definition 1 suggests a naive solution. Given $M_\times$ and $P_\times$, one can check if they satisfy the correctness property:

$$\begin{aligned} &\forall (u_1, v_1) \in I_\times,\ \forall k > 0,\ \exists sel_1,\ \cdots, sel_k : \\ &\quad T_\times(u_i, v_i, sel_i, u_{i+1}, v_{i+1}) \Rightarrow (u_{i+1}, v_{i+1}) \in P_\times. \end{aligned} \tag{6}$$

Intuitively, the property requires that for all pairs within the initial correspondence, there exists a fair path such that all pairs along the path satisfy observational equivalence. The existence of a lasso-shaped fair path can be verified by a fairness checking algorithm [15]. Nevertheless, an exponential number of pairs may exist in $I_\times$, and enumerating all those pairs is computationally intractable even when the fairness checking algorithm is incremental. Another way to deal with the quantifier alternation problem is to eliminate all the internal existential quantifiers. However, an equisatisfiable formula without existential quantifiers can be exponential in size.

C. Inductive Invariant for Equivalence modulo Stuttering

A conventional *inductive invariant* proves that a transition system always satisfies a safety property P by showing $\neg P$-states are never reachable from the initial states. We devise a new type of inductive invariant to prove that two systems are observational equivalent modulo stuttering:

Definition 2. *$Inv_\times$ is an inductive invariant modulo stuttering for $M_\times$ and $P_\times$ if it satisfies all of the following conditions:*

- $I_\times \Rightarrow Inv_\times$, *(7a)*
- $\forall s \in Inv_\times : (s \wedge T_{syn} \Rightarrow Inv'_\times) \vee (s \wedge T_{stu} \Rightarrow Inv'_\times)$, *(7b)*
- $Inv_\times \Rightarrow P_\times$. *(7c)*

Lemma 1. *M_A and M_B are observational equivalent modulo stuttering if and only if there exists an $Inv_\times$ for $M_\times$ and $P_\times$.*

Proof. Only-if part: We show the existence constructively. From Definition 1, if the two systems are equivalent modulo stuttering, there exists a $P_\times$-lasso from every state $s_1 \in I_\times$. Let $Inv_\times$ be the formula that contains exactly all pairs along these lassos.
If part: Consider an arbitrary $s_1 \in I_\times$. From (7a), $s_1 \in Inv_\times$; recursively applying condition (7b) shows that there exists an $Inv_\times$-lasso from s_1; all pairs along the lasso are $P_\times$-pairs (7c). Because a $P_\times$-lasso exists for every pair in $I_\times$, M_A and M_B are equivalent modulo stuttering. ☐

Various methods can infer inductive invariants for a transition system [10], [16]. Nevertheless, due to the *disjunction operator* in (7b), these methods cannot be applied directly for the inference of $Inv_\times$. For example, a key procedure in IC3 is to check if a clause c is relatively inductive to a frame F_i, *i.e.*, $F_i \wedge c \wedge T \Rightarrow c'$ is a tautology. It can be verified that if two clauses c_1 and c_2 are both relatively inductive to F_i, their conjunction $c_1 \wedge c_2$ is also relatively inductive to F_i. This property allows IC3 to learn new clauses incrementally while maintaining the structure of the frame sequence. However, if two clauses both meet condition (7b), their conjunction may not meet the same condition. This limitation disallows us from adopting existing inductive invariant inference algorithms to our problem.

D. The Reversed Approach

To address the above issue, we tackle the problem from the opposite direction. In specific, we switch the roles of $I_\times$ and $\neg P_\times$. Additionally, we reverse the directions of both T_{syn} and T_{stu} by switching their current states and next states, yielding T°_{syn} and T°_{stu}. Thus, the reversed inductive invariant for observational equivalence modulo stuttering can be defined as follows:

Definition 3. *$Inv^\circ_\times$ is a reversed inductive invariant modulo stuttering for $M_\times$ and $P_\times$ if it satisfies all of the following conditions:*

- $\neg P_\times \Rightarrow Inv^\circ_\times$, *(8a)*
- $\forall s_a, s_b \in Inv^\circ_\times :$
  $(s_a \wedge T^\circ_{syn} \wedge s') \wedge (s_b \wedge T^\circ_{stu} \wedge s') \Rightarrow (s' \in {Inv^\circ_\times}')$, *(8b)*
- $Inv^\circ_\times \Rightarrow \neg I_\times$. *(8c)*

Notice that there is no longer a disjunction operator in Definition 3. The next lemma states the correlation between $Inv_\times$ and $Inv^\circ_\times$.

Lemma 2. *When $Inv^\circ_\times = \neg Inv_\times$, $Inv_\times$ is an inductive invariant modulo stuttering if and only if $Inv^\circ_\times$ is a reversed inductive invariant modulo stuttering.*

Proof. (7a) (resp. (7c)) is the contrapositive statement of (8c) (resp. (8a)). The negation of (7b) is equisatisfiable to: $\exists s \in Inv_\times,\ s'_a, s'_b \in \neg Inv_\times : (s \wedge T_{syn} \wedge s'_a) \wedge (s \wedge T_{stu} \wedge s'_b)$, which in turn is equisatisfiable to the negation of (8b). Hence, (7) and (8) are equisatisfiable when $Inv^\circ_\times = \neg Inv_\times$. ☐

[1] By adding a third branch, our method can be generalized to the case that both systems can be faster than the other.

Theorem 3. *M_A and M_B are observational equivalent modulo stuttering if and only if there exists an $Inv_\times^\circ$ for $M_\times$ and $P_\times$.*

Proof. Readily follows from Lemma 1 and Lemma 2. □

Intuitively, the existence of $Inv_\times^\circ$ guarantees that a counterexample tree, whose leaf nodes are all $\neg P_\times$-states and whose root node is an $I_\times$-state, cannot exist.

Based on the above observations, we devise a method to search for a reversed inductive invariant modulo stuttering. We embed our method into IC3's general framework (Section II-C). In the remainder of this section, we highlight some key procedures we adapt from IC3. An overall description is left in the next section.

- Similar to (2), our method maintains a sequence of frames, where $F_0^\circ = \neg P_\times$ and $F_i^\circ \Rightarrow \neg I_\times$. One exception is (2b). If two states sharing the same parent state are both F_i°-states, that parent state must be an F_{i+1}°-state:

$$\begin{aligned}&\forall s_a, s_b \in F_i^\circ : \\ &\quad (s_a \wedge T_{syn}^\circ \wedge s') \wedge (s_b \wedge T_{stu}^\circ \wedge s') \Rightarrow (s' \in {F_{i+1}^\circ}'). \end{aligned} \tag{9}$$

Hence, F_i° is an over-approximation of those states which are at most i steps away from $\neg P_\times$.

- Our method extracts new proof obligations through the query:

$$SAT?\ (F_k^\circ \wedge T_{syn}^\circ \wedge I'_\times) \wedge (F_k^{\circ *} \wedge T_{stu}^\circ \wedge I'_\times). \tag{10}$$

Notice that F_k° and $F_k^{\circ *}$ represent two different sets of variables. If it is satisfiable, two new states $s \in F_k^\circ$ and $s^* \in F_k^{\circ *}$ are extracted and added to the queue of proof obligations.

- When the original IC3 discharges a proof obligation, if the query to (4) is satisfiable, a predecessor t of s is extracted and added to the priority queue. We mimic that procedure by querying

$$SAT?\ (F_i^\circ \wedge T_{syn}^\circ \wedge s') \wedge (F_i^{\circ *} \wedge T_{stu}^\circ \wedge s'). \tag{11}$$

This procedure allows our method to find candidate bad states that are more than 1 step away from $I_\times$, thus diversifying the clauses in the sequence of the frames.

- When a query to (11) is unsatisfiable, the newly produced clause $c = \neg s$ must be *relatively inductive modulo stuttering* to F_i, *i.e.*, for any state $\bar{s}$,

$$(F_i^\circ \wedge T_{syn}^\circ \wedge \bar{s}') \wedge (F_i^{\circ *} \wedge T_{stu}^\circ \wedge \bar{s}') \Rightarrow (\bar{s}' \in c'). \tag{12}$$

Hence, c can be conjoined to F_{i+1}° without violating any of the frame structure conditions including (9).

IV. Algorithm

A. The SE3 Algorithm

The core procedures of SE3 are displayed in Algorithm 1. SE3 is designed to retain several desirable features of IC3: *a)* organizing clauses in a sequence of relatively inductive frames to enable fully incremental verification (Line 18; 30); *b)* generalizing clauses to accelerate convergence (Line 29); *c)* propagating clauses to increase the chance of finding an inductive invariant (Line 16-18); and *d)* being compatible with counterexample-guided abstraction refinement workflows (Line 9, 26, 11-12). From a high-level perspective, SE3 adapts the general framework of IC3 (Section II-C) and combines it with the reversed invariant finding strategy [17] as well as the implicit abstraction technique [18]. Due to the similarity, we build our correctness proof on top of that in the IC3 paper [10]:

Theorem 4. *Upon termination,* CHECK *returns* `True` *if and only if the systems in $M_\times$ are observational equivalent modulo stuttering.*

Algorithm 1 SE3 Algorithm for Stuttering Equivalence Checking

1: **procedure** CHECK($I_\times, T_\times^\circ, P_\times$) $\rightarrow$ **bool**:
2:   **if** SAT?($I_\times \wedge \neg P_\times$) **or** SAT?($I'_\times \wedge T_\times^\circ \wedge \neg P_\times \wedge \neg P_\times^*$) **then**
3:     **return** `False` ▷ *concrete counterexample found early*
4:   $F_0^\circ \leftarrow \neg P_\times$
5:   $k \leftarrow 1,\ F_k^\circ \leftarrow \neg I_\times$
6:   **while** `True` **do**
7:     **while** SAT?($I'_\times \wedge T_\times^\circ \wedge F_k^\circ \wedge F_k^{\circ *}$) **do**
8:       $\langle s, s^* \rangle \leftarrow$ the extracted states inside F_k° and $F_k^{\circ *}$
9:       $\langle \hat{s}, \hat{s}^* \rangle \leftarrow$ the abstraction of $\langle s, s^* \rangle$ as partitions of terms
10:       **if not** (BLOCK($k, \hat{s}$) **or** BLOCK($k, \hat{s}^*$)) **then**
11:         **if** the abstract counterexample is spurious **then**
12:           refine the abstract space with more terms
13:         **else return** `False` ▷ *concrete counterexample found*
14:     $k \leftarrow k+1,\ F_k^\circ \leftarrow \neg I_\times$
15:     **for** $i \leftarrow 1$ **to** $k-1$ **do**
16:       **for** each clause $c \in F_i^\circ$ **do** ▷ *clause propagation*
17:         **if not** SAT?($\neg c' \wedge T_\times^\circ \wedge F_i^\circ \wedge F_i^{\circ *}$) **then**
18:           add c to F_{i+1}°
19:       **if** $F_i^\circ = F_{i+1}^\circ$ **then**
20:         **return** `True` ▷ *invariant found, equivalence proved*

21: ▷ *Recursive version for simplicity; actually priority-queue-based.* ◁
22: **procedure** BLOCK($i, \hat{s}$) $\rightarrow$ **bool**:
23:   **if** $i = 0$ **or** $\hat{s} \Rightarrow \neg P_\times$ **then return** `False` ▷ *bad state reached*
24:   **while** SAT?($\hat{s}' \wedge T_\times^\circ \wedge F_{i-1}^\circ \wedge F_{i-1}^{\circ *}$) **do**
25:     $\langle t, t^* \rangle \leftarrow$ the extracted states inside F_{i-1}° and $F_{i-1}^{\circ *}$
26:     $\langle \hat{t}, \hat{t}^* \rangle \leftarrow$ the abstraction of $\langle t, t^* \rangle$ as partitions of terms
27:     **if not** (BLOCK($i-1, \hat{t}$) **or** BLOCK($i-1, \hat{t}^*$)) **then**
28:       **return** `False` ▷ *blocking fails for both branches*
29:   $c \leftarrow$ the clause generalized from $\neg \hat{s}$
30:   add c to $F_1^\circ, \cdots, F_i^\circ$
31:   **return** `True` ▷ *blocking succeeds with the frames updated*

Proof. (Sketch) According to Theorem 3, the decision of observational equivalence modulo stuttering is reduced to the searching of an $Inv_\times^\circ$. When both invocations of BLOCK return `False` at Line 10, there is an abstract counterexample where *all* paths lead to $\neg P_\times$ states. If CHECK returns `False` at Line 3, a trivial counterexample is found; if it returns `False` at Line 13, a concrete counterexample corresponding to the abstract one is found. In both cases, a valid $Inv_\times^\circ$ cannot exist. On the other hand, if CHECK returns `True`, Line 20 must have been reached. This implies that two consecutive frames are identical and a valid $Inv_\times^\circ$ is found. □

In general, if the word-level models under verification reside in an infinite space, there are no complexity bounds for a model checking algorithm. However, if all variables in the models are constant-sized bit vectors, SE3 will eventually terminate because both the abstraction refinement process and the frame sequence converging process are strictly monotonic.

B. Syntax-Guided Abstraction and Refinement

An appropriate abstract space can guide the model checking algorithm to find a concise and essential inductive invariant. Our insight is that an equivalence relation can usually be expressed by *terms* within either data-intensive or control-intensive models. Besides, the clause and frame structure of IC3 allows the expression of not only one-to-one mappings between terms but also the relations among terms described by logic formulas acrossing time domains.

The SE3 algorithm maintains a set of terms that are currently used to depict the abstract space. Once a counterexample is found, it is validated through SMT queries [11]. If it is confirmed to be spurious, SE3 investigates the unsat core and adds new terms to the

set to eliminate the counterexample. SE3 prioritizes state variables and I/O variables over internal nodes and constants. If none of the above work, SE3 will also attempt to add primed variables, trying to capture the correlations between clock cycles. With this strategy, SE3 iteratively refines the granularity of its reasoning domain until an inductive invariant or a concrete counterexample is found.

V. Experimental Results

A. Experimental Setup

We implemented the SE3 algorithm in Python, using Boolector [19] as the backend SMT solver. Our implementation takes two RTL-level or gate-level Verilog designs as the inputs. It supports 2-branch (one design being no slower than the other) and 3-branch (no timing constraint) modes.

All evaluations are conducted on a Linux machine with a 3.2GHz CPU. Each instance runs on a single thread. We set a 4GiB memory limit and a 7,200-second timeout for all experiments.

In the first part of our evaluation, we compare the performance of SE3 and Kairos [8] and investigate their underlying mechanisms in a case study. We choose nuXmv [18] and AVR [20] as the backend model checkers for Kairos, because they won the first and the third place in the prestigious HWMCC'20 contest [21]. Both nuXmv and AVR are word-level safety property checkers based on implicit predicate abstraction and syntax abstraction, respectively.

For the case study, we manually write 4 RTL-level Verilog implementations of our running example. Fig. 1 shows their pseudo-code. Among those, (a) and (b) correspond to Listing 1 and Listing 2 respectively, while (c) is an alternative implementation of the running example. We intentionally add a fourth implementation, (d), which is not equivalent modulo stuttering to any of the rest implementations.

In the second part of our evaluation, we assess the capability and characteristics of SE3 in a realistic setting. We leverage a high-level synthesis (HLS) tool, Xilinx Vivado HLS, to generate pairs of non-cycle-accurate RTL designs. The HLS workflow is a combination of software compilation and hardware optimization. Because a commercial HLS tool contains almost all kinds of sequential transformations, it is an excellent source to emulate realistic design transformations and generate a variety of designs with guaranteed correctness.

We select 7 high-level hardware specifications from the HLSynth benchmark suite [22] for HLS. Our selection is based on two rules: *i)* the benchmark is a standalone module, and *ii)* at least 6 different RTL-level implementations with different timing can be generated from the benchmark using Vivado HLS. Table II provides a summary of those benchmarks. We generate 3 specifications for each benchmark with the word size set to 8, 16, and 32 bits, respectively. For each specification, we generate 6 designs with different timing, thus yielding a total of 15 pairs of designs. We utilize HLS pragmas, including pipeline, initiation interval, resource allocation, latency, unroll, flatten, merge, partition, balance, etc., to control a design's timing. Eventually, we obtained a total of 315 pairs as our test cases. Because nuXmv is almost always faster than AVR when paired with Kairos, we only compare with *Kairos/nuXmv* in the second part.

B. Case Study

Table I compares the performances of Kairos and SE3 on the test cases shown in Fig. 1. The first two test cases check designs that are equivalent modulo stuttering. Even though Kairos uses word-level model checkers as its backend, its performance deteriorates quickly as the word size grows. On the contrary, the execution time of SE3 grows more slowly.

We believe that Kairos is over-conservative when aligning two sequences. As illustrated in Fig. 2(a), Kairos enforces both designs to execute when none of them or both of them reach a $valid$ state; it stalls the faster design if exactly one of them reaches a $valid$ state. Such an alignment pattern hinders the underlying model checker from finding a concise inductive invariant.

Test Case	Algorithm	3bit	4bit	5bit	6bit	8bit	16bit	32bit
(a) vs. (b) equiv.	*Kairos/nuXmv*	0.41	4.13	163.3	–	–	–	–
	Kairos/AVR	18.17	323.4	–	–	–	–	–
	SE3	0.04	0.05	0.06	0.06	0.07	0.19	0.51
(a) vs. (c) equiv.	*Kairos/nuXmv*	0.44	3.25	315.0	–	–	–	–
	Kairos/AVR	10.9	96.06	–	–	–	–	–
	SE3	0.23	0.29	0.44	0.80	1.34	4.42	13.63
(a) vs. (d) non-equiv.	*Kairos/nuXmv*	0.05	0.07	0.08	0.11	0.13	0.30	0.74
	Kairos/AVR	1.02	4.07	17.99	38.96	239.1	294.1	570.2
	SE3	1.32	1.77	2.53	3.27	4.77	8.24	36.75

TABLE I: A comparison of execution time *(s)* under different word sizes.

The last test case in Table I checks a pair of non-equivalent designs. *Kairos/nuXmv* turns out to be faster than SE3 in finding a counterexample. It is because Kairos prunes out all alignment patterns except one. In this regard, Kairos and SE3 are complementary to each other. A hybrid sequential equivalence checking engine can run the two algorithms in parallel to achieve optimal performance.

if $x > y$ **then** $x \leftarrow x - y$
else if $y > x$ **then** $y \leftarrow y - x$
(a) gcd_A

if $x > y$ **then** $x \leftarrow x - y$
else swap x and y
(b) gcd_B

$msb \leftarrow$ the highest bit of y
if $msb = 0$ **and** $x > y * 2$ **then**
  $x \leftarrow x - y * 2$
else if $x > y$ **then** $x \leftarrow x - y$
else swap x and y
(c) gcd_C

▷ *Possible overflow in* $y * 2$ ◁
if $x > y * 2$ **then**
  $x \leftarrow x - y * 2$
else if $x > y$ **then** $x \leftarrow x - y$
else swap x and y
(d) gcd_D (incorrect)

Fig. 1: Loop bodies of 4 implementations of the gcd running example.

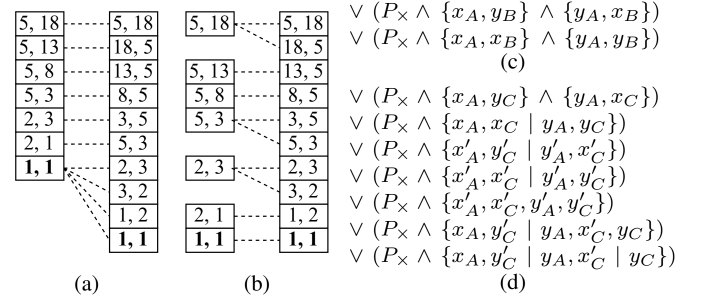

Fig. 2: Alignments of state sequences starting from $(5, 18)$ for gcd_A vs. gcd_B by (a) Kairos and (b) SE3 (**bold** states are the $valid$ states); the inductive invariants in partition assignment representation found by SE3 for (c) gcd_A vs. gcd_B and (d) gcd_A vs. gcd_C.

Fig. 2(c) and 2(d) showcase the inductive invariants for gcd_A vs. gcd_B and gcd_A vs. gcd_C, respectively. Because gcd_A vs. gcd_C is harder to prove, SE3 introduced primed terms during refinements to capture the relations between the current and the next state variables.

C. Experimental Results

The outcomes of our experiments are shown in Fig. 3. Either in the 3-branch mode or the 2-branch mode, SE3 can solve more equivalent test cases than Kairos within any period. We also observe that both the 2-branch and the 3-branch modes can solve some extra test cases than the other modes within the same period. Specifically, the 2-branch mode is more efficient when a design is always faster, while the 3-branch mode can cope with the general situation. Hence, a

Name	Description	# Nodes	# Regs
gcd	GCD Algorithm	60-115	6-11
euclid	Alternative GCD Algorithm	58-109	6-11
counter	Bidirectional Counter with Limit	96-164	6-14
diffeq	Differential Equation Solver	86-170	6-15
barcode	Barcode Reader	93-187	8-18
ellipf	Fifth Order Elliptical Wave Filter	135-210	11-21
kalman	Kalman Filter	157-229	8-26

TABLE II: A summary of the benchmarks. The number of nodes and registers are counted by terms.

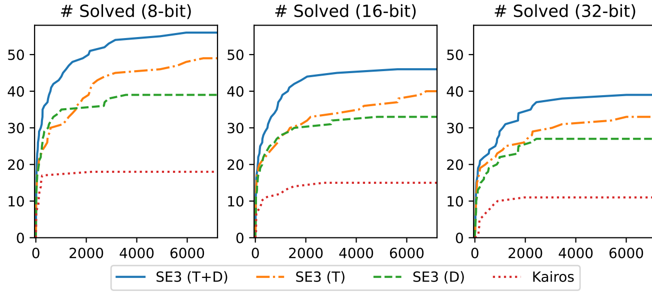

Fig. 3: The number of solved instances over time *(s)* under different word sizes. T refers to the 3-branch mode and D refers to the 2-branch mode.

combination of the two modes running in parallel can solve the largest number of test cases.

We notice that Kairos is closer to SE3 in terms of execution time on two benchmarks: *barcode* and *ellipf*. These benchmarks produce outputs periodically, triggering the $valid$ signal frequently. Kairos can leverage this additional information to accelerate its computation.

Finally, we analyze the statistics during SE3's execution. Thanks to the close integration of the model checking algorithm and syntax abstraction, the number of total frames and SMT queries are almost unchanged with respect to the word size. The average time spent per SMT query moderately increases as the word size grows, and this accounts for the trend shown in Table I.

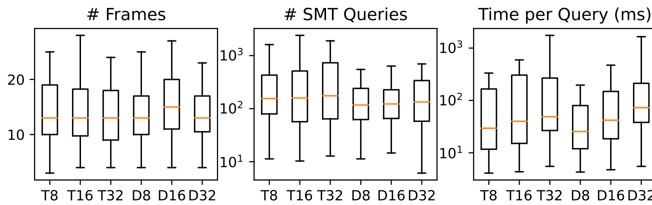

Fig. 4: Statistics of SE3 under different word sizes.

VI. Related Work

Mitering is the standard technique to check if two designs are combinationally equivalent. A miter composes the designs under verification by connecting their corresponding inputs. Then an SAT/SMT solver is queried to check if there exists an input pattern for the two designs to produce different output patterns. Chauhan *et. al.* attempts to reduce sequential equivalence checking problems to combinational ones [4]. They iteratively unroll both designs until their periods are both 1. The validity of this approach is based on two assumptions: both designs have fixed periods, and a latch mapping is provided by the user. These assumptions may not hold after significant transformations.

Most algorithms on sequential equivalence checking [5]–[7] utilize structural similarities between the designs to verify their equivalence. They either use SAT/BDD sweeping to identify and compress equivalent internal nodes and combine them into equivalent classes, or use random simulation and counterexample-guided refinement to partition the classes. The algorithms terminate when all corresponding output nodes from both designs are proven to be equivalent. Otherwise, they launch a sequence of rewriting and retiming steps and repeat the whole process. However, a fixed one-to-one mapping of internal nodes from both designs may not exist. Moreover, rewriting and retiming rely heavily on heuristics, which are incomplete methods. In comparison, our proposed method enables high flexibility to express the correlations between internal signals. Additionally, it is based on model checking and guarantees soundness and completeness.

VII. Conclusion

In this paper, we propose SE3, an efficient algorithm for non-cycle-accurate equivalence checking. SE3 first reduces the sequential equivalence checking problem to the problem of checking the alignablity of observable sequences. It then solves the problem by finding reversed inductive invariants on a syntax abstracted space. Experimental results confirm the effectiveness and correctness of SE3.

References

[1] S. Dai, G. Liu, R. Zhao, and Z. Zhang, “Enabling adaptive loop pipelining in high-level synthesis,” in *ACSSC'17*. IEEE, pp. 131–135.
[2] H. Peng *et al.*, “A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining,” in *DAC'22*, pp. 1135–1140.
[3] N. Sharma, G. Hasteer, and V. Krishnaswamy, “Sequential equivalence checking for rtl models. eetimes now,” 2006.
[4] P. Chauhan *et al.*, “Non-cycle-accurate sequential equivalence checking,” in *DAC'09*. IEEE, pp. 460–465.
[5] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, “Solver technology for system-level to rtl equivalence checking,” in *DATE'09*, pp. 196–201.
[6] J. Baumgartner *et al.*, “Scalable sequential equivalence checking across arbitrary design transformations,” in *ICCAD'06*. IEEE, pp. 259–266.
[7] C. Van Eijk, “Sequential equivalence checking without state space traversal,” in *DATE'98*. IEEE, pp. 618–623.
[8] L. Piccolboni *et al.*, “Kairos: Incremental verification in high-level synthesis through latency-insensitive design,” in *FMCAD'19*.
[9] C. Pixley, “Introduction to a computational theory and implementation of sequential hardware equivalence,” in *ICCAD'90*. Springer, pp. 54–64.
[10] A. R. Bradley, “Sat-based model checking without unrolling,” in *VM-CAI'11*. Springer, pp. 70–87.
[11] A. Goel and K. Sakallah, “Model checking of verilog rtl using ic3 with syntax-guided abstraction,” in *NASA Formal Methods Symposium*. Springer, 2019, pp. 166–185.
[12] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.
[13] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Parameter synthesis with ic3,” in *FMCAD'13*. IEEE, pp. 165–168.
[14] A. R. Bradley *et al.*, “An incremental approach to model checking progress properties,” in *FMCAD'11*. IEEE, pp. 144–153.
[15] K. Claessen and N. Sörensson, “A liveness checking algorithm that counts,” in *FMCAD'12*. IEEE, pp. 52–59.
[16] K. L. McMillan, “Interpolation and sat-based model checking,” in *CAV'03*. Springer, pp. 1–13.
[17] T. Seufert, C. Scholl, D. Groe, and R. Drechsler, “Sequential verification using reverse pdr.” in *MBMV*, 2017, pp. 79–90.
[18] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “Ic3 modulo theories via implicit predicate abstraction,” in *TACAS'14*. Springer, pp. 46–61.
[19] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, “Btor2, btormc and boolector 3.0,” in *ICCAD'18*. Springer, pp. 587–595.
[20] A. Goel and K. Sakallah, “Avr: abstractly verifying reachability,” in *TACAS'20*. Springer, pp. 413–422.
[21] M. Preiner, A. Biere, and N. Froleyks, “Hardware model checking competition 2020,” 2020.
[22] P. R. Panda and N. D. Dutt, “1995 high level synthesis design repository,” in *ISSS'95*, pp. 170–174.