Optimal Stateless Model Checking based on View-equivalence

***Abstract*—Partitioning program executions into equivalence classes is central to efficient stateless model checking of concurrent programs. An equivalence relation drives the partitioning, and prior works have investigated various such relations toward minimizing the set of equivalence classes. This work introduces a novel *view-equivalence* partitioning relation that relies on the values read by memory accesses. Two program executions are view-equivalent if (i) they have the same set of `read` memory accesses and (ii) the `read` memory accesses read the same values. For any input program, view-equivalence is at least as coarse as any existing equivalence relation and, thus, induces the least number of equivalence classes. This paper also presents a stateless model checker based on view-equivalence, called ViEqui, and shows that ViEqui is sound, complete, and optimal under view-equivalence. ViEqui is implemented for C/C++ input programs over the Nidhugg tool. We test its correctness over 16000+ litmus tests and compare its performance against prior stateless model checkers on challenging benchmarks. Our experiments reveal that ViEqui performs over 10x faster in ∼23% of tests and times out in ∼29% lesser tests than the fastest existing technique.**

***Index Terms*—stateless model checking, concurrency, coarse equivalence, view-equivalence**

I. Introduction

The interleaving model of reasoning about concurrent programs suffers from the combinatorial explosion in the interleavings of thread instructions. Stateless model checking addresses the combinatorial explosion problem by exploring a reduced state graph of interleaving sequences with respect to a safety property ψ. Stateless model checkers (SMCs) partition the program executions into *equivalence classes* based on an *equivalence relation* such that either all executions of an equivalence class satisfy ψ or all satisfy $\neg\psi$. As a consequence, an SMC may analyze any one representative execution from each equivalence class.

Evidently, minimizing the set of equivalence classes positively impacts the model checking effort. The challenge remains in defining a suitable equivalence relation that partitions the execution sequences, such that, (i) all sequences in a partition behave similarly to a property ψ (i.e. satisfy ψ or satisfy $\neg\psi$), and (ii) there exists an equivalence class whose sequences satisfy ψ (if feasible) and an equivalence class whose sequences satisfy $\neg\psi$ (if feasible).

Several prior works have investigated various definitions of equivalence which present a range of coarse equivalence relations that depend on program attributes such as the order of racing events (shared memory accesses) [1], [2], [3], the order of *observable* racing events [4], [5], the *reads-from* relation [6], [7], [8], [9], and the causality of `read` events [10], [11].

This paper proposes a novel equivalence relation called *view-equivalence* (refer to §II) that relies only on the values of `read` events. Two executions are *view-equivalent* if they have the same set of `read` events and each `read` event reads the same value. For any input program, view-equivalence is at least as coarse as any existing equivalence relation, and thus, induces the least number of equivalence classes.

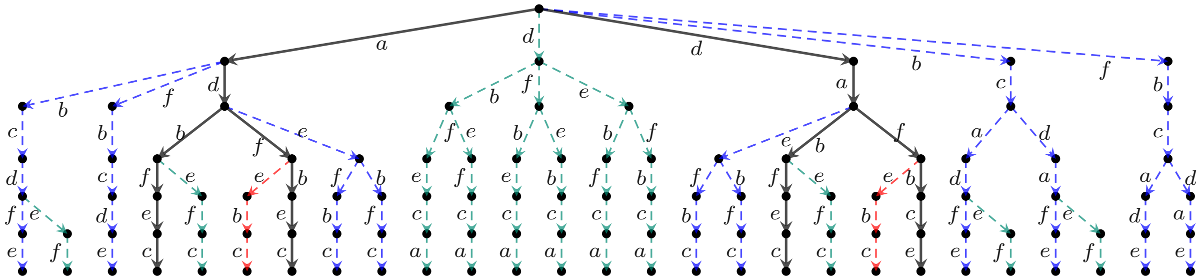
This paper also presents an SMC called ViEqui (pronounced as `vee.key`), which partitions program executions using the proposed view-equivalence relation under sequential consistency. ViEqui analyses representative executions from the view-equivalence classes of an input program for safety property (assert) violations.

The set of equivalence classes is not known to an SMC a priori, thus, SMCs compute equivalence classes on-the-fly. Accurate on-the-fly computation of equivalence classes could realize exploration of exactly one execution sequence from each equivalence class, called an *optimal exploration*. An SMC that explores optimally is called an *optimal SMC*. As a third contribution, this paper shows that ViEqui is *sound*, *complete*, and *optimal* under view-equivalence partitioning (refer to §VI).

Similar to other techniques [1], [5], ViEqui computes a set of event sequences at each state of exploration, such that extending the exploration prefix with the sequences results in the exploration of previously unanalyzed equivalence classes. Existing SMCs fundamentally rely on the order of occurrence of events in an execution to define an equivalence class and explore new equivalence classes by altering the order of events of an explored sequence. However, various orders of occurrence on events may correspond to the same view-equivalence class, even those where a `read` event reads from different `write` events, if they write the same value. It is non-trivial to associate sequences with an unordered set of events and their corresponding values that define a view-equivalence class. As a result, the problem of generating the required set of sequences at a state is acutely challenging under view-equivalence, and a simple modification of occurrence order (as in [1], [5]) may not result in a different equivalence class.

More generally, representations of equivalence classes that exploit the commutativity of concurrent events, such as *ample sets* [12], *persistent sets* [13], and *source sets* [14], become unusable under view-equivalence. Similar representations for previously explored equivalence classes, such as *done sets* [3] and *sleep sets* (over events [13] or event sequences [15]) cannot effectively represent view-equivalence classes.

Note that a view-equivalence class is defined by a set of `read` events and their corresponding values. Accordingly, this



classical (*Mazurkiewicz*) equivalence: blue, green, red dashed sequences and **black** solid sequences
observed racing-pairs equivalence: blue, red dashed sequences and **black** solid sequences
reads-from and reads-value-from equivalence: red dashed sequences and **black** solid sequences
view-equivalence: **black** solid sequences

Fig. 1: Equivalence classes of program in Figure 2 under various equivalence relations.

paper proposes a novel representation for view-equivalence classes as a sum-of-product formula (refer to §V), where a term of the formula is a (read event, read value) pair. For instance, $(a, 0) \vee (b, 1)$ represents the set of view-equivalence classes where either the read a reads the values 0, or read b reads the value 1; and, $(a, 0) \wedge (b, 1)$ represents a single view-equivalence class where a reads 0 and b reads 1 (assuming only two read events in the input program). Such a notion coarsens the representation of an equivalence class by disassociating it with event sequences.

Given a view-equivalence class (as a sum-of-product formula) to be explored from a state, ViEqui computes an event sequence at the state, that extends the execution prefix with the read events and read values corresponding to the class. ViEqui recognizes write events of the required values, and determines if the writes can be read by the read events, coherently under sequential consistency (using an operator $\oplus$, refer to §VI). Lastly, ViEqui schedules the coherent sequence for execution only if it does not break optimality (computed using an operator $\uplus$, refer to §VI).

ViEqui is implemented over *Nidhugg* [16] for concurrent C/C++ programs. We test the implementation on 16154 litmus tests of multi-threaded C programs and compare its performance against other optimal SMCs [1], [5] and an SMC that considers values to define equivalence [10] (refer to §VII). Our experiments show that ViEqui performs verification over 10x faster in ∼23% tests and times out in ∼29% lesser tests than the fastest existing technique (∼30% and ∼46% respectively on an average across techniques [1], [5], [10]).

II. Equivalence of Program Executions

Classically, equivalence between program executions is defined on *Mazurkiewicz traces* [17] formed using *racing events* (pairs of events, accessing the same shared object, where at least one event is a write) as *Mazurkiewicz* dependent events [1], [2], [3], [18], [19]. Program executions with the same order of occurrence on racing events are considered equivalent (we refer to this equivalence relation as the *classical* equivalence relation). Consider program $\mathcal{P}1$ in Figure 2. Notation $W(o, v)$ represents a write to a shared object o with value v and, $R(o)$ represents a read of a shared object o. The parallel bars (‖) represent the parallel composition of events

$$\begin{array}{c}
\text{Initially } x = 0, y = 0 \\
a\text{: } W(x,1) \;\Big\Vert\; \begin{array}{l} b\text{: } R(y) \\ c\text{: } W(x,1) \end{array} \;\Big\Vert\; \begin{array}{l} d\text{: } R(x) \\ e\text{: } W(y,1) \end{array} \;\Big\Vert\; f\text{: } W(y,1)
\end{array}$$

Fig. 2: $\mathcal{P}1$. concurrent program with repeating write values

of different program threads, and 'a...f' are event labels. The program has three pairs of racing events on each shared object (x and y); thus, under the classical equivalence relation, the program has 27 distinct equivalence classes (computed as $3! \times 3! - 9$ combinations that cannot be realized due to a cycle in the execution). The equivalence classes for the example program are shown in Figure 1 (through representative executions). Every execution depicted in Figure 1 (including the dashed and the solid sequences) represents a distinct valid classical equivalence class.

The classical notion of equivalence is sound for detecting data races and safety property (assert) violations. Recent works have explored coarser notions of equivalence that pivot on assert violations. The coarser notions are discussed below.

Equivalence on observed races [4], [5]. Program executions with the same order of occurrence on *observed* racing event pairs (event pairs that contain a read event or are followed by one) are considered equivalent. Program $\mathcal{P}1$ has 16 such equivalence classes, represented by all solid and dashed sequences, except the green dashed sequences, in Figure 1. Consider the sequences $a.b.c.d.f.e$ and $a.b.c.d.e.f$ (left-most two sequences in Figure 1) that vary in the order of writes e and f (we refer to events by their labels), which are not observed. As a result, the executions are equivalent.

Reads-from equivalence [6], [7], [9]. Two executions are equivalent if they have the same set of read events that read from the same write events. Each of the two read events (b, d) in program $\mathcal{P}1$ (Figure 2) can read from three write events (including the initial write event). Thus, in total, there are 6 equivalence classes under the reads-from relation (represented by red dashed and black solid sequences in Figure 1).

Reads-value-from equivalence [7], [10]. Two executions are equivalent if they have the same set of read events that read the same value and maintain the same *causal-ordering*. The two read events of the program $\mathcal{P}1$ (Figure 2) can read two values each (i.e. 0 and 1). The read b is causally ordered before d when d reads 1 from c, but not when d reads 1 from a. Thus, d (similarly b) reads 1 in two equivalence classes and reads 0 in a third class. As a result, the program in Figure 2 has the same set of equivalence classes under this relation as under reads-from equivalence.

View-equivalence

In this work, we propose a coarse notion of equivalence called view-equivalence that relies only on the values of `read` events. Two executions are equivalent if they have the same set of `read` events that read the same value.

Given sequences τ_1 and τ_2, let $\tau_1 \sim \tau_2$ represent that τ_1 and τ_2 are *view-equivalent*, formally defined as,

Definition 1. (view-equivalence)

$$\tau_1 \sim \tau_2 \text{ iff } \mathcal{E}^{\mathbb{R}}_{\tau_1} = \mathcal{E}^{\mathbb{R}}_{\tau_2} \wedge \forall e_r \in \mathcal{E}^{\mathbb{R}}_{\tau_1}, val_{[\tau_1]}(e_r) = val_{[\tau_2]}(e_r)$$

The view-equivalence relation partitions the program executions in *view-equivalence classes*.

The program $\mathcal{P}1$ in Figure 2 has exactly four view-equivalence classes corresponding to the combinations of values 0 and 1 for the two `read` events. The equivalence classes are represented by black solid sequences in Figure 1 (each dashed sequence can be partitioned with the equivalence class of one of the black solid sequences).

Theorem 1. For a given input program, view-equivalence is at least as coarse as any existing equivalence relation.

View-equivalence may significantly save on the model checking effort (as exhibited by Figure 1). At the same time, the applicability of view-equivalence is not reduced in the context of assert violations compared to the other coarse equivalence relations. Intuitively, every observable location and every branching location in the input program, such as the SSA phi nodes, output statements, and assert statements, rely on the values of the corresponding `read` events. Hence, by considering all combinations of `read` values, a model checker explores every branch, output of the program, and outcome of an assert statement.

III. Overview of ViEqui

ViEqui takes a multi-threaded program as input and explores (executes and analyzes) its view-equivalence classes under sequential-consistency. ViEqui transitions from a state of exploration to the next by executing a program event. At each state, ViEqui computes a set of values that can be read by the `read` events. Subsequently, ViEqui schedules and examines executions where the computed values are read.

The primary objective of the technique is to compute a relevant set of sequences from each state $s_{[\tau]}$ (the state reached after exploring a sequence τ), called *next-steps* (*Nxt*$(s_{[\tau]})$), such that, on extending the sequences in *Nxt*$(s_{[\tau]})$, ViEqui

(a) explores all equivalence classes reachable from $s_{[\tau]}$, and
(b) explores no redundant executions (executions that are *view-equivalent* to another explored execution).

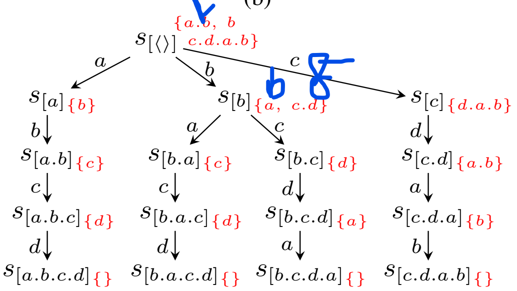
Initially $x = 0, y = 0$

a: $W(x, 1)$	b: $R(x)$	c: $R(y)$ d: $R(x)$

(a)

next-steps at initial state (*Nxt*$(s_{[\langle\rangle]})$):
$\{a.b.c.d,\ b.a.c.d,\ c.d.a.b,\ c.d.b.a\}$,
$\{b.a,\ c.d,\ a.b\}$,
$\{b,\ a.b,\ c.d.a.b\}$ *etc.*

(b)



(c)

Fig. 3: $\mathcal{P}2$. (a) input program, (b) sample set of valid next-steps, (c) exploration by ViEqui

Consider the program $\mathcal{P}2$ in Figure 3(a). Figure 3(b) shows three sets of sequences that represent valid sets of next-steps from the initial state ($s_{[\langle\rangle]}$). ViEqui computes one such set of sequences for each state of exploration on-the-fly.

Figure 3(c) shows the exploration of the program by ViEqui. ViEqui starts the exploration from the initial state $s_{[\langle\rangle]}$ where the events a, b, c are enabled (or available for execution). From the enabled events, ViEqui computes that the `read` b can read two values (i.e. 0 and 1). The technique then *eagerly* (without executing) computes *Nxt*$(s_{[\langle\rangle]}) = \{b, a.b\}$ where b can read the values 0 and 1, respectively. This eager computation of *Nxt*$(s_{[\tau]})$ is called *forward-analysis*.

Forward-analysis examines only the enabled events at a state $s_{[\tau]}$, hence, computing *Nxt*$(s_{[\tau]})$ precisely for all events is not always possible with forward-analysis. For example, the event d is not available for analysis at $s_{[\langle\rangle]}$. To address such a scenario, ViEqui performs analysis for such events after they are enabled. We refer to this analysis as *backward-analysis*, since it requires examining the prefix of an execution. Consider the state $s_{[a.b.c]}$ where the `read` event d is enabled after the execution of c . The event d can only read the value 1 in this execution. However, d can also read the value 0 (initial value) in some execution. ViEqui computes (i) an explored state where the value 0 can indeed be read by d (i.e. $s_{[\langle\rangle]}$), (ii) a prefix sequence that would enable d after $s_{[\langle\rangle]}$ without introducing another `write` before d, (i.e. $c.d$), and (iii) a sequence that will lead to the intended read from $s_{[\langle\rangle]}$ while still maintaining the value for b (i.e. $c.d.a.b$). The sequence $c.d.a.b$ is added to *Nxt*$(s_{[\langle\rangle]})$. Note that, maintaining the value of b is essential for optimality (formally discussed in §V).

After exploring an entire execution (i.e. $a.b.c.d$), ViEqui backtracks to the state that has unexplored next-steps (i.e. $s_{[\langle\rangle]}$) and proceeds to explore b from $s_{[\langle\rangle]}$. A similar backward-analysis as in the first execution leads to the computation of sequence $b.c.d$, where, b is already explored from $s_{[\langle\rangle]}$, hence, $c.d$ is added to $\mathit{Nxt}(s_{[b]})$. The technique stops after exploring all next-steps computed by forward- and backward analyses.

A key feature of ViEqui that prevents redundant explorations and ensures termination of analysis is the representation of explored view-equivalence classes as a sum-of-product formula called *skip*. Skip is associated with next-steps and guides the respective explorations on what view-equivalence classes are to be *skipped* or omitted for analyses. On computation of the next-step $c.d.a.b$ in Figure 3(c), ViEqui adds a term of $(d, 1)$ to the skip associated with $c.d.a.b$. The term signifies that view-equivalence classes corresponding to the value 1 for read d have been explored with some other execution, and no further next-steps are to be added from the execution sequence $c.d.a.b$. The computation of skip is formally discussed in §V.

IV. Preliminaries

Concurrent model. Consider an acyclic multi-thread program P with a finite set of program threads. Each thread P_i has deterministic computations and terminating executions. The threads access a single shared memory.

Program events. A thread executes a sequence of *events*, where an event e is composed of a sequence of internal actions on local objects followed by a read or write action on a shared object. Accordingly, an event of a thread P_i is a tuple $\langle P_i, a, o\rangle$ where a represents an action $\in \{\texttt{read}, \texttt{write}\}$ on a shared object o from a finite set of objects accessed by P. We use $obj(e)$ to represent the object of an event e. Let $\mathcal{E}$ represent the set of events of P. Notations $\mathcal{E}^{\mathbb{W}}$ and $\mathcal{E}^{\mathbb{R}}$ represent the write and read events of P, such that, $\mathcal{E}^{\mathbb{W}} \cup \mathcal{E}^{\mathbb{R}} = \mathcal{E}$ and $\mathcal{E}^{\mathbb{W}} \cap \mathcal{E}^{\mathbb{R}} = \emptyset$. Further, for each shared object o we consider a special *initial event* $(\mathbb{I}_o)$.

Event sequences and program executions. A sequence of program events, which is maximal (i.e. there are no enabled events after the sequence) represents a program execution. A *sequence* may refer to either non-maximal sequences or maximal sequences of events. Given a sequence τ, $\mathcal{E}_\tau$, $\mathcal{E}^{\mathbb{W}}_\tau$ and $\mathcal{E}^{\mathbb{R}}_\tau$ represent respectively the sets of events, writes and reads occurring in τ. In a sequence τ, $val_{[\tau]}(e)$ represents the value of an event e from a finite set of program values $\mathcal{V}$. Given $e_r \in \mathcal{E}^{\mathbb{R}}_\tau$, e_r reads the value of the latest write of $obj(e_r)$, in the prefix of a sequence τ, up to e_r (represented by $\mathit{lastW}_{[\tau]}(e_r)$). However, if there does not exist a write event in the prefix of τ up to e_r then $\mathit{lastW}_{[\tau]}(e_r) = \mathbb{I}_{obj(e_r)}$.

Exploration states. The state reached after executing a sequence τ is denoted as $s_{[\tau]}$, where a state is defined as the valuation of shared and local objects and program counters of each thread. The set of enabled events at $s_{[\tau]}$ is represented as $\mathit{En}(s_{[\tau]})$. The execution of an event from a thread P_i enables the next event in *program-order* from P_i, thus, the enabled set contains the *next* event from each thread. This implies that, when a write event e_w is enabled in a sequence τ, $val_{[\tau]}(e_w)$ is already computed for τ and available as a constant.

Notation on event sets and sequences. A sequence τ is extended by an event e or a sequence τ' as $\tau.e$ (respectively $\tau.\tau'$). An empty sequence is represented by $\langle\rangle$. We use $e \in A$ to represent that an event e is contained in A, where A can be a set or a sequence. Given sequences τ_1, τ_2, we use $\tau_1 \sqcap \tau_2$ to represent the longest sequence τ, that is a subsequence of τ_1 and τ_2, and $\tau_1 - \tau_2$ represents the remaining sequence of τ_1 after removing its subsequence τ_2.

V. Key analyses for computing view-equivalence

The set of view-equivalence classes are not known a priori, thus, ViEqui computes the set during exploration of program executions. ViEqui computes the set of sequences $\mathit{Nxt}(s_{[\tau]})$ that can extend the execution prefix τ to program executions, while ensuring, (n1) $\mathit{Nxt}(s_{[\tau]})$ can extend to a representative execution for each view-equivalence class *reachable* from $s_{[\tau]}$, and (n2) no two sequences in $\mathit{Nxt}(s_{[\tau]})$ can extend to execution sequences representing the same view-equivalence class.

Let Π represent the set of view-equivalence classes for the input program. Consider $\mathit{rch}_{[\tau']}(\tau'') \subseteq \Pi$, a set of view-equivalence classes reachable from $s_{[\tau'.\tau'']}$. The set contains view-equivalence classes that have a representative execution with $\tau'.\tau''$ as a prefix. The conditions (n1) and (n2) for constructing $\mathit{Nxt}(s_{[\tau]})$ are formally stated as:

$$\bigcup_{\tau' \in \mathit{Nxt}(s_{[\tau]})} \mathit{rch}_{[\tau]}(\tau') = \begin{cases} \mathit{rch}_{[\langle\rangle]}(\tau) & \text{if } \tau \neq \langle\rangle \\ \Pi & \text{if } \tau = \langle\rangle \end{cases} \qquad (\texttt{n1})$$

$$\forall \tau_i, \tau_j \in \mathit{Nxt}(s_{[\tau]}), \tau_i \neq \tau_j, \mathit{rch}_{[\tau]}(\tau_i) \cap \mathit{rch}_{[\tau]}(\tau_j) = \emptyset \quad (\texttt{n2})$$

Consider the state $s_{[b]}$ in Figure 3(c); $\mathit{Nxt}(s_{[b]}) = \{a, c.d\}$; $\mathit{rch}_{[b]}(a)$ contains a single view-equivalence class, corresponding to $b{=}0, d{=}1$ and, $\mathit{rch}_{[b]}(c.d)$ also contains a single view-equivalence class, corresponding to $b{=}0, d{=}0$. Hence, $\mathit{rch}_{[b]}(a) \cup \mathit{rch}_{[b]}(c.d) = \mathit{rch}_{[\langle\rangle]}(b)$ (satisfying (n1)), and $\mathit{rch}_{[b]}(a) \cap \mathit{rch}_{[b]}(c.d) = \emptyset$ (satisfying (n2)).

To facilitate the construction of the required $\mathit{Nxt}(s_{[\tau]})$, ViEqui performs the following analyses: (i) compute context of read; (ii) succinctly summarize the explored view-equivalence classes; and, (iii) add next-steps at a state without introducing redundancies (using a special operator $\uplus$). The details of the analyses are as follows.

Computing context of read. A read event may read the same value in various view-equivalence classes. As a consequence, ViEqui computes a context of a read (where the context is interpreted in terms of the other reads and their values), during forward- and backward-analysis.

Consider the read f at $s_{[b.d.e]}$ (in Figure 4(b)). Since c is not enabled in τ_3, value 0 cannot be explored for f with forward-analysis in τ_3. Hence, backward-analysis is performed for value 0 and read f from τ_3. Observe that, there exist two view-equivalence classes where f reads 0 i.e. the classes corresponding to $b{=}0, f{=}0$ and $b{=}1, f{=}0$ (which is explored by τ_1). Hence, backward-analysis computes a sequence where f reads 0 'in the context that b reads 0' and may form either of the sequences $e.f.b$ or $e.b.f$ or $b.e.f$ as the next-step.

Succinct summary of explored view-equivalence classes. During exploration of the input program, it is necessary to maintain knowledge of the previously examined view-equivalence classes to avoid redundant exploration and ensure termination (where, examined refers to classes that are already explored or would eventually be explored because a respective next-step has already been formed).

ViEqui represents previously examined view-equivalence classes as a sum-of-product formula called *skip*, where a term of the formula is a (`read` event, `read` value) pair. If valuation of a term $(r, v) = \top$ in a formula S results in valuation of S $= \top$, represented as $(r, v) \models \mathsf{S}$, then ViEqui does not compute a next-step where r reads v. The initial value of skip (as with any sum-of-product formula) is $\bot$.

View-equivalence classes are in essence a combination of `read` events and their corresponding values. Therefore, such a representation is meaningful under view-equivalence.

Next-step containers. ViEqui associates skip with the next-steps at each state forming pairs of (next-step, skip) called *next-step containers*. The set of next-step containers at a state $s_{[\tau]}$ is denoted by $\overline{Nxt}(s_{[\tau]})$. Further, for $c \in \overline{Nxt}(s_{[\tau]})$, $ns(c)$ and $sk(c)$ project the next-step and skip of c, respectively.

Consider again the backward-analysis at $s_{[b.d.e]}$ (in τ_3, Figure 4(b)). Assume that backward-analysis computed the next-step $e.f.b$. The computed next-step is associated with the skip $(f, 1) \vee (b, 1)$, where $(f, 1)$ corresponds to the current value of f and $(b, 1)$ is the skip associated with the context of the read. The skip thus captures the previously known skip terms, i.e. $(b, 1)$, and the current value, i.e. $(f, 1)$, signifying that they are examined. The resulting next-step container is $(e.f.b, (f, 1) \vee (b, 1))$ (shown in green in Figure 4(b)).

Further, consider the sequence $\tau_4 = b.e.f.d.a$, where $val_{[\tau_4]}(f) = 0$. The `read` event f may read the value 1 from d, however, $(f, 1) \models sk(f.d, (f, 1))$ at $s_{[b.e]}$. Thus, ViEqui does not add a next-step for `read` f and value 1 from τ_4.

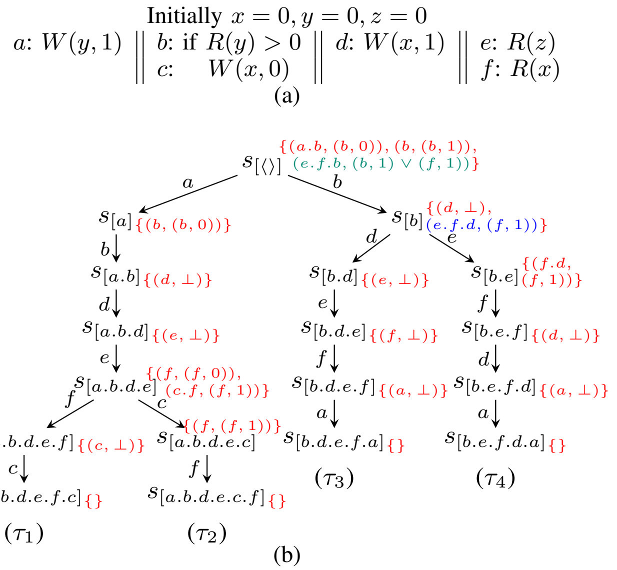
If $(r, v) \models sk(c)$, then it implies that r has already read v in the context of the other `reads` and their values in $ns(c)$; for example $(f, 1) \models sk(e.f.b, (f, 1) \vee (b, 1))$ and the value 1 is read for f in the context of b=0 (in τ_3).

Addition of next-steps using $\uplus$. ViEqui defines an operator $\uplus$ that computes redundancies in the set of classes reachable from a fresh next-step against those reachable from next-steps already at a state. The operator adds a fresh next-step at a state only in the absence of any such redundancies. Formally, consider the addition of c to $\overline{Nxt}(s_{[\tau]})$, computed as,

Definition 2. (consistent-union, $\uplus$)

While computing $\overline{Nxt}(s_{[\tau]}) \uplus c$, consider $c' \in \overline{Nxt}(s_{[\tau]})$ s.t.
if $ns(c) \sim ns(c')$ then $Nxt(s_{[\tau]}) \setminus c' \cup (ns(c'), sk(c') \vee sk(c))$
if $ns(c).\tau_1 \sim ns(c').\tau_2$ then $Nxt(s_{[\tau]}) \setminus c' \uplus (ns(c') \sqcap ns(c), sk(c') \wedge sk(c))$ (for some sequences τ_1, τ_2)

Special case. Since, ViEqui computes $\overline{Nxt}(s_{[\tau]})$ while exploring program executions, the $c' \in \overline{Nxt}(s_{[\tau]})$ may have already executed (and cannot be effectively removed from $Nxt(s_{[\tau]})$). To handle such a scenario ViEqui performs additional computation (refer to Appendix A). For instance, if $ns(c')$ is being currently explored then the sequence $ns(c) - ns(c')$ is added at $s_{[\tau.ns(c')]}$ (using $\uplus$). The additional computation performs exploration of c in the current exploration itself, ensuring that exploration of c is not missed out.



```
Initially x = 0, y = 0, z = 0
a: W(y,1) || b: if R(y) > 0 || d: W(x,1) || e: R(z)
          || c:    W(x,0)   ||           || f: R(x)
```

(a)

$s_{[\langle\rangle]}$ $\{(a.b, (b, 0)), (b, (b, 1)), (e.f.b, (b, 1) \vee (f, 1))\}$
a
b
$s_{[a]}$ $\{(b, (b, 0))\}$
$s_{[b]}$ $\{(d, \bot), (e.f.d, (f, 1))\}$
b
d
e
$s_{[a.b]}$ $\{(d, \bot)\}$
$s_{[b.d]}$ $\{(e, \bot)\}$
$s_{[b.e]}$ $\{(f.d, (f, 1))\}$
d
e
f
$s_{[a.b.d]}$ $\{(e, \bot)\}$
$s_{[b.d.e]}$ $\{(f, \bot)\}$
$s_{[b.e.f]}$ $\{(d, \bot)\}$
e
f
d
$s_{[a.b.d.e]}$ $\{(f, (f, 0)), (c.f, (f, 1))\}$
$s_{[b.d.e.f]}$ $\{(a, \bot)\}$
$s_{[b.e.f.d]}$ $\{(a, \bot)\}$
f
c
a
a
$s_{[a.b.d.e.f]}$ $\{(c, \bot)\}$
$s_{[a.b.d.e.c]}$ $\{(f, (f, 1))\}$
$s_{[b.d.e.f.a]}$ $\{\}$
$s_{[b.e.f.d.a]}$ $\{\}$
c
f
$s_{[a.b.d.e.f.c]}$ $\{\}$
$s_{[a.b.d.e.c.f]}$ $\{\}$
(τ_1)
(τ_2)
(τ_3)
(τ_4)

(b)

Fig. 4: $\mathcal{P}3$. (a) input program, (b) exploration by ViEqui

Consider $c = (e.f.b, (f, 1) \vee (b, 1))$ (shown in green in Figure 4(b)). There exist $\tau_1 = \langle\rangle$ and $\tau_2 = e.f$ s.t. $e.f.b.\tau_1 \sim b.\tau_2$. Since, b is currently being explored, ViEqui computes $e.f.b - b$ $= e.f$, that is added to $s_{[b]}$ in the context of $(d, \bot)$. Thus, $(e.f.d, (f, 1))$ is added to $s_{[b]}$ (shown in blue in Figure 4(b)).

VI. ViEqui ALGORITHM

Consider the notations formally presented in Figure 5 and explained below, that are used by the ViEqui algorithm.

Event relations and related sequences. Given a sequence τ, $<_\tau$ represents the relation *occurs-before* on the events of τ, $\rightarrow_\tau^{\text{po}}$ represents the *program-order* on the events of a thread, and $\rightarrow_\tau^{\text{rf}}$ represents the *reads-from* relation. Two events are *causally-ordered* ($\rightarrow_\tau^{\text{co}}$) if they are ordered by the transitive closure of $\rightarrow_\tau^{\text{po}} \cup \rightarrow_\tau^{\text{rf}}$; and a *causal join* ($\oplus$) joins two sequences while preserving causal-ordering. An *enabling sequence* of $e \in \mathcal{E}_\tau$ ($eseq_{[\tau]}(e)$), is a smallest subsequence of τ that enables e. Intuitively, the events that are causally-ordered before e in τ enable e in τ; for example, consider the sequence τ_1 in Figure 4(b), $eseq_{[\tau_1]}(c) = a.b.c$.

Operations for next-steps. Notations $currNS(s_{[\tau]})$ and $currSK(s_{[\tau]})$ represent the next-step, and its corresponding skip, being explored from $s_{[\tau]}$; on first visit to $s_{[\tau]}$, $currNS(s_{[\tau]}) = \langle\rangle$ and $currSK(s_{[\tau]}) = \bot$. Let $ns\text{-}state_{[\tau]}(e)$ represent the state where the next-step containing e starts exploration; for an initial event $\mathbb{I}_o$, $ns\text{-}state_{[\tau]}(\mathbb{I}_o) = s_{[\langle\rangle]}$.

Consider τ_2 in Figure 4(b), $currNS(s_{[a.b.d.e]}) = c.f$ and $currSK(s_{[a.b.d.e]}) = (f, 1)$. Further, $ns\text{-}state_{[\tau_2]}(f) = s_{[a.b.d.e]}$.

Notations $currNS(s_{[\tau]})$ and $currSK(s_{[\tau]})$ return the context for creating a next-step; $ns\text{-}state_{[\tau]}(e)$ returns the starting state of e's next-step and is used as the state to add a fresh next-step on e. The choice of $ns\text{-}state_{[\tau]}(e)$ ensures that a fresh next-

$e_1 <_\tau e_2$	$\triangleq$	e_1 occurs before e_2 in τ
$e_1 \rightarrow_{\tau}^{\text{po}} e_2$	$\triangleq$	e_1 occurs before e_2 in the same thread
$e_w \rightarrow_{\tau}^{\text{rf}} e_r$	$\triangleq$	$e_w, e_r \in \mathcal{E}_\tau$ and $e_w = \mathit{lastW}_{[\tau]}(e_r)$
$\mathit{eseq}_{[\tau]}(e)$	$\triangleq$	smallest subsequence of τ s.t. $\forall e' \in \mathit{eseq}_{[\tau]}(e), e'{=}e \vee \exists e_{po} \rightarrow_{\tau}^{\text{po}} e$ s.t. $e' = e_{po} \vee e' \rightarrow_{\tau}^{\text{co}} e_{po}$ (note, $\mathit{eseq}_{[\tau]}(\mathbb{I}_o) = \langle\rangle$)
$\mathit{currNS}(s_{[\tau]})$	$\triangleq$	next-step being explored at $s_{[\tau]}$
$\mathit{currSK}(s_{[\tau]})$	$\triangleq$	skip of $\mathit{currNS}(\tau)$
$\mathit{ns\text{-}state}_{[\tau]}(e)$	$\triangleq$	smallest prefix $\tau'$of τ s.t. $e{\in}\mathit{currNS}(s_{[\tau']})$
$\mathit{readable}_{[\tau]}(e_r)$	$\triangleq$	$\{e_w \in \mathcal{E}_\tau^{\mathbb{W}} \mid \mathit{nseq}_{[\tau',\tau]}(e_w, e_r) {\neq} \langle\rangle \wedge \mathit{obj}(e_w){=}\mathit{obj}(e_r)\}$ ($\tau'{=}\mathit{currNS}(\mathit{ns\text{-}state}_{[\tau]}(e_w))$, if $e_w <_\tau e_r$, or, $\mathit{currNS}(\mathit{ns\text{-}state}_{[\tau]}(e_r))$, if $e_r <_\tau e_w$)
$\mathit{before}_{[\tau]}(e_r)$	$\triangleq$	$\subseteq \mathit{readable}_{[\tau]}(e_r)$ s.t. $e_w <_\tau e_r$
$\mathit{after}_{[\tau]}(e_r)$	$\triangleq$	$\subseteq \mathit{readable}_{[\tau]}(e_r)$ s.t. $e_r <_\tau e_w$
$\mathit{disjunct}_{[\tau]}(e_r, W)$	$\triangleq$	$\bigvee_{e_w \in W}(e_r, \mathit{val}_{[\tau]}(e_w))$
$e_1 \rightarrow_{\tau}^{\text{co}} e_2$	$\triangleq$	$(e_1, e_2) \in$ transitive closure of $(\rightarrow_{\tau}^{\text{po}} \cup \rightarrow_{\tau}^{\text{rf}})$
$\tau_1 \oplus \tau_2$	$\triangleq$	if $\exists$ (irreflexive) τ s.t. $\mathcal{E}_\tau = \mathcal{E}_{\tau_1} \cup \mathcal{E}_{\tau_2}$ and $\rightarrow_{\tau}^{\text{co}} = \rightarrow_{\tau_1}^{\text{co}} \cup \rightarrow_{\tau_2}^{\text{co}}$, then $\tau_1 \oplus \tau_2 = \tau$, otherwise $\tau_1 \oplus \tau_2 = \langle\rangle$.
$\mathit{nseq}_{[\tau',\tau]}(e_w, e_r)$	$\triangleq$	$\mathit{eseq}_{[\tau'']}(e_w) \oplus \mathit{eseq}_{[\tau'']}(e_r) \oplus e_w.e_r \oplus \mathit{currNS}(s_{[\tau']})$ (where, $\tau = \tau'.\tau''$)
$\mathit{unique}_{[\tau]}(E)$	$\triangleq$	$\forall e, e' \in E$ s.t. $\mathit{val}_{[\tau]}(e) = \mathit{val}_{[\tau]}(e')$ either $e \in \mathit{unique}_{[\tau]}(E)$ or $e' \in \mathit{unique}_{[\tau]}(E)$ but not both.
$\mathit{dup}_{[\tau]}(E, e)$	$\triangleq$	$\{e' \in E \mid \mathit{val}_{[\tau]}(e') = \mathit{val}_{[\tau]}(e)\}$
$\mathit{co\text{-}en}_{[\tau]}(e_r)$	$\triangleq$	$\{e_w \in \mathcal{E}^{\mathbb{W}} \cap \mathit{En}(s_{[\tau]}) \mid \mathit{obj}(e_w) = \mathit{obj}(e_r)\}$ if $e_r \in \mathit{En}(s_{[\tau]})$, and $\emptyset$ otherwise.
$\mathit{done}_{[\tau]}(e_r)$	$\subseteq$	$\{e_w \in \mathcal{E}_\tau^{\mathbb{W}} \cup \{\mathbb{I}_{obj(e_r)}\} \mid \mathit{obj}(e_w) = \mathit{obj}(e_r)\}$ s.t. $\forall e_w \in \mathit{done}_{[\tau]}(e_r)$ where $\mathit{val}_{[\tau]}(e_w){=}v$, (i) $v = \mathit{val}_{[\tau]}(\mathit{lastW}_{[\tau]}(e_r))$, or (ii) $(e_r, v) \models \mathit{currSK}(\mathit{ns\text{-}state}_{[\tau]}(e_r))$, or (iii) $e_w \in \mathit{currNS}(\mathit{ns\text{-}state}_{[\tau]}(e_r))$.

Fig. 5: Notations used by the ViEqui algorithm (Algorithm 1)

step is not added in the middle of another next-step as such a scenario may cause an improper computation of reachability.

Construction of next-step. Given $\tau = \tau'.\tau''$, a next-step at $s_{[\tau']}$ where e_r reads e_w ($\mathit{nseq}_{[\tau',\tau]}(e_w, e_r)$) is constructed such that, it enables e_w and e_r, ensures e_r reads e_w and establishes the context by including $\mathit{currNS}(s_{[\tau]})$.

For example, the next-step $e.f.b$ (shown in green in Figure 4(b)) is constructed as $\langle\rangle \oplus e.f \oplus \langle\rangle.f \oplus b$. Further, consider the next-step $c.d.a.b$ at $s_{[\langle\rangle]}$ in Figure 3(c). The next-step is constructed as $\langle\rangle \oplus c.d \oplus \langle\rangle.d \oplus a.b$, where $\oplus$ ensures a occurs after d as otherwise the resulting sequence would have $a \rightarrow^{\text{rf}} d \Rightarrow a \rightarrow^{\text{co}} d \notin \rightarrow_{c.d}^{\text{co}} \cup \rightarrow_{d}^{\text{co}} \cup \rightarrow_{a.b}^{\text{co}}$ (invalid by definition of $\oplus$).

As a result, $\oplus$ ensures that the read value for which the next-step is being formed, along with the read values of the enabling sequences and context, are preserved in the resulting next-step; otherwise, $\oplus$ returns $\langle\rangle$ and no next-step is formed.

Sets on events. Given a set of events E, $\mathit{unique}_{[\tau]}(E) \subseteq E$ represents a set of events unique in value, and $\mathit{dup}_{[\tau]}(E, e) \subseteq E$ returns a set of events that share the value of e.

Given a `read` event e_r, consider the following sets on `writes` of the same object: the set of *co-enabled* `writes` ($\mathit{co\text{-}en}_{[\tau]}(e_r)$); the set of `writes` explored before ($\mathit{before}_{[\tau]}(e_r)$) and after ($\mathit{after}_{[\tau]}(e_r)$) e_r in τ, such that a next-step can be formed where e_r reads from the `write`; and, the set of `writes` that are *done* ($\mathit{done}_{[\tau]}(e_r)$), which includes (i) `writes` that share the memory value, (ii) `writes` already examined, and (iii) `writes` in the context.

Others. Let $\mathit{disjunct}_{[\tau]}(e_r, W)$ represent the disjunction of terms of a `read` e_r and the values of `writes` in *W*.

A. Forward-analysis and backward-analysis

Using the notations presented in Figure 5, forward-analysis and backward-analysis are formally defined as functions *fwd*, *bkwdWR* and *bkwdRW*.

Forward-analysis. Forward-analysis is formally presented as Function *fwd*. The function computes a set (unique in values) of enabled writes (*W*) that are not in $\mathit{done}_{[\tau]}(e_r)$ (line 2). Forward-analysis is performed on an enabled `read` (e_r) (shown in Algorithm 1), hence, the set *W* contains `writes` of the same object as e_r that are co-enabled with e_r. The function *fwd* then computes next-step containers for the memory value (lines 3-4) and the co-enabled `writes` in *W* (lines 5-7). Note that, for each $e_w \in W$ (and the memory value) the corresponding next-step adds a term in skip for all other values in *W* (lines 4, 6-7), representing that the other values in *W* have also been examined.

Consider the program $\mathcal{P}4$ in Figure 6, function *fwd* recognizes values 0, 1, 2 for the `read` e from the enabled `writes` and the memory value. Initially skip = $\bot$, thus the set W = $\mathit{unique}_{[\langle\rangle]}(\{b, c, d\})$ = $\{b, d\}$ or $\{c, d\}$. Assuming W = $\{b, d\}$, line 4 computes $(e, (e, 1) \vee (e, 2))$; and lines 5-7 compute $(b.e, (e, 0) \vee (e, 2))$ and $(d.e, (e, 0) \vee (e, 1))$.

Backward analysis. Backward-analysis is performed after exploring a maximal sequence, for `read` and `write` events that are not co-enabled. The backward-analysis for a `read` e_r and a `write` e_w where $e_w <_\tau e_r$ is presented as function *bkwdWR*, and where $e_r <_\tau e_w$ as function *bkwdRW*.

Function bkwdWR. The function is performed for a `read` e_r, to read from `writes` explored before e_r in τ whose values are not already examined (line 2). Consider the sequence τ_3 in

```
1 Function $fwd(explored\ sequence\ \tau,\ \texttt{read}\ event\ e_r)$:
2     $W := unique_{[\tau]}(co\text{-}en_{[\tau]}(e_r) \setminus done_{[\tau]}(e_r))$
3     if $(e_r, v_{\mathsf{mem}}) \not\models currSK(s_{[\tau]})$ then
4         $\overline{Nxt}(s_{[\tau]}) \uplus= (nseq_{[\tau,\tau]}(mem, e_r), disjunct_{[\tau]}(e_r, W))$
5     forall $e_w \in W$ do
6         $W' := (W \cup mem) \setminus e_w$
7         $\overline{Nxt}(s_{[\tau]}) \uplus= (nseq_{[\tau,\tau]}(e_w, e_r),\ disjunct_{[\tau]}(e_r, W'))$
```

let mem = $lastW_{[\tau]}(e_r)$; $v_{mem} = val_{[\tau]}(mem)$ (value in memory)

```
1 Function $bkwdWR(sequence\ \tau,\ \texttt{read}\ event\ e_r)$:
2     $W := (before_{[\tau]}(e_r) \cup \{\mathbb{I}_o\}) \setminus done_{[\tau]}(e_r)$
3     $W' := after_{[\tau]}(e_r)$
4     forall $e_w \in W$; do $s_{[\tau_w]}$ = $ns\text{-}state_{[\tau]}(e_w)$
5         $W := W \setminus dup_{[\tau]}(W, e_w)$; $W' := W' \setminus dup_{[\tau]}(W', e_w)$
6         $new\text{-}skip := currSK(s_{[\tau_r]}) \vee currSK(s_{[\tau_w]}) \vee$
            $disjunct_{[\tau]}(e_r, W \cup mem \cup W')$
7         $\overline{Nxt}(s_{[\tau_w]}) \uplus= (nseq_{[\tau_w,\tau]}(e_w, e_r), new\text{-}skip)$
8         forall $c \in \overline{Nxt}(s_{[\tau_r]})$ do
9             $\overline{Nxt}(s_{[\tau_r]}) \setminus c \cup (ns(c), sk(c) \vee (e_r, val_{[\tau]}(e_w)))$
```

let mem = $lastW_{[\tau]}(e_r)$; $o = obj(e_r)$; $s_{[\tau_r]}$ = $ns\text{-}state_{[\tau]}(e_r)$

```
1 Function $bkwdRW(sequence\ \tau,\ \texttt{read}\ event\ e_r)$:
2     $W := after_{[\tau]}(e_r) \setminus done_{[\tau]}(e_r)$
3     $N := \mathcal{V} \mapsto \emptyset$
4     forall $e_w \in W$ do $v = val_{[\tau]}(e_w)$
5         $new\text{-}skip := currSK(s_{[\tau_r]}) \vee (e_r, val_{[\tau]}(e_r))$
6         $N[v]\ \cup=\ (nseq_{[\tau_r,\tau]}(e_w, e_r),\ new\text{-}skip)$
7     forall $v \in N$ and $c_1, c_2 \in N[v]$ do
8         if $rch_{[\tau_r]}(ns(c_1)) \subseteq rch_{[\tau_r]}(ns(c_2))$ then $N[v] \setminus= c_1$
9     forall $v \in N$ and $c \in N[v]$ do $\overline{Nxt}(s_{[\tau_r]})\ \uplus= c$
```

let $s_{[\tau_r]} := ns\text{-}state_{[\tau]}(e_r)$

Figure 4(b). The `read` f reads 1 from d in τ_3, hence the set of `writes` explored before f, that are not *done*, W = $\{\mathbb{I}_x\}$.

For each e_w in W, the function computes the prefix τ_w for adding a next-step, that is, the prefix where the next-step containing e_w started exploration (line 4). The next-step is computed using $nseq_{[\tau_w,\tau]}(e_w, e_r)$ (line 7) and the associated skip contains terms from previously known skip (associated with next-steps of e_r and e_w), current value of e_r, and values of other `writes` that can be read by e_r (line 6).

Consider again the `read` f in τ_3 of Figure 4(b) and the computed W = $\{\mathbb{I}_x\}$; τ_w corresponding to $\mathbb{I}_x$ = $\langle\rangle$. The next-step is computed as $\langle\rangle \oplus e.f \oplus \langle\rangle.f \oplus b$ and *new-skip* = $\bot \vee (b, 1) \vee (f, 1)$; where, $currSK(s_{[\tau_r]})$ = $\bot$, $currSK(s_{[\tau_w]})$ = $(b, 1)$, and $disjunct_{[\tau]}(e_r, W \cup mem \cup W')$ = $(f, 1)$.

Finally, the value of e_w is recorded as already examined for e_r by discarding other `writes` of same value (line 5) and adding a term of $(e_r, val_{[\tau]}(e_w))$ to all next-step containers at $s_{[\tau_r]}$ (lines 8-9). This ensures that no other event of the same value as e_w, from no other sequence extending after τ_r, would perform backward-analysis for the value of e_w.

Function bkwdRW. The function is performed for a `read` e_r,

Initially $x = 0$

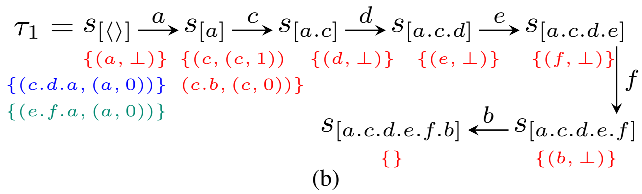
a: $W(x, 0)$ || b: $W(x, 1)$ || c: $W(x, 1)$ || d: $W(x, 2)$ || e: $R(x)$

Fig. 6: $\mathcal{P}4$. example of forward-analysis

Initially $x = 0, y = 0, z = 0$

a: $R(x)$ || c: $R(y)$ || e: $R(z)$
b: $W(y, 1)$ || d: $W(x, 1)$ || f: $W(x, 1)$

(a)


$\tau_1 = s_{[\langle\rangle]} \xrightarrow{a} s_{[a]} \xrightarrow{c} s_{[a.c]} \xrightarrow{d} s_{[a.c.d]} \xrightarrow{e} s_{[a.c.d.e]}$
$\{(a, \bot)\}$
$\{(c, (c, 1))$
$(c.b, (c, 0))\}$
$\{(d, \bot)\}$
$\{(e, \bot)\}$
$\{(f, \bot)\}$
$\{(c.d.a, (a, 0))\}$
$\{(e.f.a, (a, 0))\}$
f
$s_{[a.c.d.e.f.b]} \xleftarrow{b} s_{[a.c.d.e.f]}$
$\{\}$
$\{(b, \bot)\}$

(b)

Fig. 7: $\mathcal{P}5$. example of backward-analysis

to read from `writes` (W) explored after e_r in τ, whose values are not already examined (line 2). For each `write` in W, the next-step is computed using $nseq_{[\tau_r,\tau]}(e_w, e_r)$ (line 6) (τ_r is the prefix where the next-step is added, that is, the prefix where the next-step containing e_r started exploration). The associated skip contains terms from previously known skip (of the next-step of e_r), and the current value of e_r (line 5).

The analysis of *bkwdRW* is called on a `read` e_r for various execution suffixes extending from $s_{[\tau_r]}$. To optimally choose a `write` event for a value in such a scenario, *bkwdRW* does not add the computed next-steps directly to $\overline{Nxt}(s_{[\tau_r]})$. The computed containers whose reachable view-equivalence classes can also be reached from another container are omitted (lines 7-8) and the remaining containers are added (using $\uplus$) to $\overline{Nxt}(s_{[\tau_r]})$ (line 9).

Consider the program $\mathcal{P}5$ in Figure 7(a) and its execution τ_1 in Figure 7(b); for the `read` a, W = $\{d, f\}$. The function computes $c_1 = (c.d.a, (a, 0))$ and $c_2 = (e.f.a, (a, 0))$ corresponding to the value 1 for a (shown in blue and green respectively). The view-equivalence class corresponding to a=1, c=0 is reachable from both c_1 and c_2, while the view-equivalence class corresponding to a=1, c=1 is reachable only from c_2. Thus, $rch_{[\langle\rangle]}(c.d.a) \subset rch_{[\langle\rangle]}(e.f.a)$. Hence, c_1 is omitted and c_2 is added to $\overline{Nxt}(s_{[\langle\rangle]})$.

B. ViEqui Algorithm (Algorithm 1)

The algorithm takes an explored sequence (τ) and a previously computed next-step to be explored (N). If there are no enabled events at $s_{[\tau]}$, then the algorithm has explored a maximal sequence (line 2). As a final step, the algorithm performs backward-analysis (line 3). If there are enabled events then the algorithm continues to explore and add next-steps (lines 5-13). If a next-step is being explored (i.e. $N \neq \langle\rangle$), then the algorithm executes the next event in N[1](line 6); otherwise (i.e. N = $\langle\rangle$), the algorithm chooses an enabled

[1] Algorithm 1 does not consider $\overline{Nxt}(s_{[\tau]})$ for all $s_{[\tau]}$ where $N \neq \langle\rangle$, however, for analyzing the conditions (`n1`) and (`n2`) in §V consider $Nxt(s_{[\tau]})$ = N.

Algorithm 1: ViEqui algorithm (**Initially** `Explore(⟨⟩,⟨⟩)`)

```
1  Function Explore(explored sequence $\tau$, next-step to explore $N$):
2      if $En(s_{[\tau]}) = \emptyset$ then            /* maximal sequence explored */
3          forall $e_r \in \mathcal{E}^{\mathbb{R}}_{\tau}$ do  $bkwdWR(\tau, e_r)$ ; $bkwdRW(\tau, e_r)$
4          return                        /* do backward-analysis and return */
5      if $N \neq \langle\rangle$ then                /* explore next event in $N$ */
6          $Explore(\tau.N{:}hd,\ N{:}tl)$ ; return
7      if $\exists e_r \in \mathcal{E}^{\mathbb{R}}$ s.t. $|co\text{-}en_{[\tau]}(e_r)| > 0$ then
8          $fwd(\tau, e_r)$          /* forward-analysis possible on $e_r$ */
9      else                             /* explore any enabled event */
10         $nexte := pickAny(En(s_{[\tau]}))$
11         $\overline{Nxt}(s_{[\tau]}) := (\langle\rangle.nexte, \bot)$
12     forall $c \in \overline{Nxt}(s_{[\tau]})$ do     /* explore all next-steps */
13         $Explore(\tau.ns(c){:}hd,\ ns(c){:}tl)$
```

where, for a sequence $\tau = e_1.e_2...e_n$, $\tau{:}hd = e_1$ and $\tau{:}tl = e_2...e_n$,

event to proceed (lines 7-11). The algorithm first detects an enabled `read` with co-enabled `writes`, for feasible forward-analysis (line 7-8). If there does not exist such a `read` then any enabled event is selected to proceed (lines 10-11). Finally, all next-steps computed by forward- and backward-analysis are explored from $s_{[\tau]}$ (lines 12-13).

Let E be the set of executions explored by ViEqui. Given $\tau \in E$ let $\tau \in \pi$, where $\pi \in \Pi$, denote that the execution τ represents the view-equivalence class π.

Theorem 2. (*soundness*) $\forall \tau \in E, \exists \pi \in \Pi$ s.t. $\tau \in \pi$.

Proof. An SMC can explore only enabled events thus, ViEqui is sound if $\forall s_{[\tau]}$ (exploration states), $\forall n \in Nxt(s_{[\tau]}) = \tau_1.e.\tau_2$, $e \in En(s_{[\tau.\tau_1]})$. If n is formed by forward-analysis then $e \in En(s_{[\tau.\tau_1]})$ (by definition). Consider n is formed by backward-analysis. Events in *eseq* and *currNS* are enabled by definition $\Rightarrow$ if $e \notin En(s_{[\tau.\tau_1]})$ then $\oplus$ forms a sequence where e is not enabled $\Rightarrow \rightarrow^{co}_{n} \not\subseteq$ causal order on operands of $\oplus \Rightarrow$ result of $\oplus = \langle\rangle \Rightarrow n$ was not formed by backward analysis. □

Theorem 3. (*completeness*) $\forall \pi \in \Pi, \exists \tau \in E$ s.t. $\tau \in \pi$.

Proof. Consider a prefix τ' such that a `read` e_r can read the value v after τ'. Let $obj(e_r) = o$. Let $\tau \in E$ s.t. τ' is prefix of τ. Assume, $(a1)$ e_r is enabled in $\tau \Rightarrow \exists \tau''$ s.t. $\tau'.\tau''.e_r$ is a prefix of τ, and $(a2)$ assume $\exists e_w \in \mathcal{E}^{\mathbb{W}}_{\tau} \cup \mathbb{I}_o$ s.t. $val_{[\tau]}(e_w) = v$. Consider a next-step n (to be formed) s.t. $val_{[n]}(e_r) = v$ and consider the absence of skip formulas. There exist three cases for e_w, (i) $e_w \in \mathcal{E}_{\tau'.\tau''} \cup \mathbb{I}_o$, or (ii) $e_w \in En(s_{[\tau'.\tau'']})$, or (iii) not cases (i) and (ii).

In case (i), if $e_w = lastW_{[\tau]}(e_r)$ and in case (ii), forward-analysis adds n to $Nxt(s_{[\tau'.\tau'']})$. In case (i), where $e_w \neq lastW_{[\tau]}(e_r)$, e_r can read $v \Rightarrow nseq_{[\tau_w,\tau]}(e_w, e_r) \neq \langle\rangle$ (where, $\tau_w = ns\text{-}state_{[\tau]}(e_w)$), thus, *bkwdWR* forms n and τ' can be extended to τ. In case (iii), e_r can read $v \Rightarrow nseq_{[\tau'',\tau]}(e_w, e_r) \neq \langle\rangle$, thus, $e_r <_{\tau} e_w$, *bkwdRW* forms n and τ' can be extended to τ.

Hence, in the absence of skip, for each `read` e_r and its value v readable after τ' a relevant next-step is formed. *inf(1)*

Consider again a prefix τ' s.t. e_r can read v after τ'. Assume $\nexists\ \tau \in E$ where τ' is a prefix of τ and $val_{[\tau]}(e_r)$.

inf(1) $\Rightarrow$ forall next-step containers c formed after τ', either (i) $(e_r, v) \models sk(c) \Rightarrow \exists$ another sequence where e_r reads v; or (ii) $e_r \in ns(c) \wedge val_{[ns(c)]}(e_r) \neq v$ (this contradicts *inf(1)*).

Lastly, since every (`read`, value) pair readable after τ' is read, it implies that the assumptions (a1) and (a2) always hold. □

Theorem 4. (*optimality*) $\nexists \tau_1, \tau_2 \in E$ $(\tau_1 \neq \tau_2)$ s.t. $\tau_1 \sim \tau_2$.

Proof. Let $\tau' \triangleleft \tau''$ represent that $\exists \tau'_1$ s.t. $\tau'.\tau'_1 \sim \tau''$.

Consider a state $s_{[\tau']}$ and $n_1, n_2 \in Nxt(s_{[\tau']})$.

If n_1 and n_2 are added by forward-analysis then $rch_{[\tau']}(n_1) \cap rch_{[\tau']}(n_2) = \emptyset$ (by definition). If n_1 is added by backward-analysis when $n_2 \in Nxt(s_{[\tau']})$ then $rch_{[\tau']}(n_2) \not\subseteq rch_{[\tau']}(n_1)$ (by definition of $\uplus$), and $rch_{[\tau']}(n_1) \not\subseteq rch_{[\tau']}(n_2)$, since $\neg n_1 \triangleleft n_2$ (because if $n_1 \triangleleft n_2$ then $\exists n'_2 \in Nxt(s_{[\tau']})$, used as $currNS(s_{[\tau']})$ to form n_1, s.t. $n'_2 \triangleleft n_2$ which violates rules of $\uplus$ computation). Hence, next-steps at a state are not redundant.

Further, consider states $s_{[\tau']}$, $s_{[\tau'']}$ and $n_1 \in Nxt(s_{[\tau']})$, $n_2 \in Nxt(s_{[\tau'']})$ s.t. $rch_{[\tau']}(n_1) \cap rch_{[\tau'']}(n_2) \neq \emptyset. \Rightarrow \tau' \sim \tau''$ $\Rightarrow$ they represent the same execution (since, next-steps at a state are not redundant) Hence, next-steps at different states are also not redundant. □

C. Complexity analysis

Since, ViEqui is optimal, each complete exploration can have at most $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ maximal sequences, the maximum number of next-steps is also $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$.

Time complexity. The worst-case time complexity of forward-analysis is $\mathscr{F} = \mathcal{O}(|\mathcal{T}|^2 + \mathtt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$, (where, $\mathcal{T}$ the set of program threads or the number of enabled events) and of backward-analyses is $\mathscr{B} = \mathcal{O}(|\mathcal{E}^{\mathbb{W}}|.(|\tau|^3 + |\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|))$. In the worst case, length of a sequence, $|\tau| = |\mathcal{E}|$.

Algorithm 1 computes forward- and backward-analyses for each `read` event, thus, its complexity is $\mathcal{O}(|\mathcal{E}^{\mathbb{R}}|.(\mathscr{F} + \mathscr{B}))$.

Space complexity. In the worst-case, the $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ next-steps are formed at $s_{[\langle\rangle]}$. However, then the size of next-steps at remaining states are $(|\mathcal{E}|-1), (|\mathcal{E}|-2), ..., 1$. Thus, for each maximal sequence the total size of next-steps is $|\mathcal{E}|.(|\mathcal{E}|+1)/2$. Given $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ maximal sequences, the worst-case space complexity of Algorithm 1 is $\mathcal{O}(|\mathcal{E}|^2.|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|})$.

VII. Implementation and Results

Implementation details. ViEqui technique is implemented in `C++` over *Nidhugg* tool [16] for `C/C++` input programs[2]. The input program is instrumented using LLVM to recognize newly enabled events dynamically. A *runtime engine* launches a new process to execute the instrumented program for every maximal sequence. At each state of exploration, the runtime engine is instructed on the next event to be executed by a *scheduler* that carries out the steps of Algorithm 1.

Experimental setup. The experiments are conducted on an Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz with

[2] ViEqui artefact is anonymously available at: https://www.dropbox.com/sh/ld34pj8setdkzb8/AABObLd6nxC8iJAcCeCwD-sda?dl=0

TABLE I: Litmus Tests

Total tests: 16154 #classical equivalence classes: between 1 to 3392
#view-equivalence classes: between 1 to 504

Category	#tests	Avg.		Total		#sound
		Seq	Time	Seq	Time	+opt
No violation	8091	10.15	0.02s	82124	171.71	8091
Has violation	8063	1.00	0.017	8066	137.87	8063

(Avg./Total) #Seq: Average/total number of sequences of tests.
(Avg./Total) Time: Average/total time of analysis of tests (over 5 runs).
#sound+opt: Number of tests verified sound and optimal.

32GB RAM and 32 cores running Ubuntu 18.04.1 LTS and LLVM 6.0.0. We perform a comparative study against two optimal verification techniques namely ODPOR [1] (optimal under classical equivalence), and ODPOR-with-observers (obs-ODPOR) [5] (optimal under equivalence based on observed races). We also compare against a non-optimal technique that uses `read` values to define equivalence called RVF-SMC [10] (based on reads-value-from equivalence).

Litmus Testing. ViEqui is tested on 16154 litmus tests of multi-threaded C programs, with a focus on, (i) reporting feasible assert-violations, (ii) soundness, and (iii) optimality. We detect soundness fail by unsuccessful program runs (since unsuccessful runs imply computation of incoherent next-steps) and optimality fail by comparing maximal sequences using Definition 1. The litmus tests, consist of tests borrowed from [20] (8058 tests), borrowed tests modified by negating the assert condition (8058 tests), and synthesized tests (38 tests), that is, a total of 16154 tests. The litmus tests consist of 8091 tests that do not violate an assert condition in any program run (category 'No violation'), and 8063 tests that violate an assert condition in some run (category 'Has violation').

The result of litmus testing is summarized in Table I.

Performance analysis. The techniques ODPOR, obs-ODPOR, RVF-SMC and ViEqui are tested on multi-threaded benchmarks borrowed from SV-comp [21], SCTBench [22] and previous works [5], [20]. The performance is measured on three aspects, (i) the time of analysis, (ii) scalability, and (iii) the number of maximal sequences explored. The time of analysis is recorded over five runs, and scalability is measured by the highest configuration of a benchmark that can be verified within a timeout of analysis (To), set at 1800 seconds. The results of the experiments are shown in Tables II and III[3]. Table II compares the performance on benchmarks where the assert condition is not violated in any execution. For such benchmarks, the techniques explore the entire set of equivalence classes and provide a proof of correctness for the input program. Table III compares performance on benchmarks with assert violation. For such benchmarks, the techniques report the assert violation and halt the exploration after detecting the first assert violation.

The columns '#Seq' represent the number of sequences explored by the techniques; for ODPOR, obs-ODPOR and ViEqui, this represents the number of equivalence classes under the respective equivalence relations, however, for RVF-SMC, the number includes redundant and incomplete explorations as well (since RVF-SMC is not optimal). The columns 'Time' represent the time of analysis .

TABLE II: Benchmarking (benchmarks with no assert violation)

test		ODPOR		obs-ODPOR		RVF-SMC		ViEqui	
ID	benchmark	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time
1	pgsql(5,5)	781	0.72	781	0.70	19900	3.06	781	0.73
2	pgsql(6,7)	55987	68.57	55987	77.51	2292077	654.66	55987	183.77
3	pgsql(7,7)	137257	171.45	137257	199.18	5356580	1620.66	137257	933.77
4	mnbsx(100)	To	-	To	-	101	0.99	1	0.08
5	mnbsx(500)	To	-	To	-	501	162.84	1	2.80
6	unvrf(5,5)	14400	2.74	14400	3.13	68890	11.70	14400	198.61
7	unvrf(5,10)	14400	2.98	14400	3.31	70890	12.76	14400	201.80
8	unvrf(6,5)	518400	110.60	518400	129.32	2625944	699.47	To	-
9	rd-co(10)	To	1800.00	12431	3.10	11	0.01	7	0.02
10	rd-co(50)	To	-	To	1800.00	11	0.02	7	0.03
11	rd-co(1000)	To	-	To	-	11	0.11	7	3.59
12	co1(20)	To	-	To	-	8060	14.14	7240	4.93
13	co1(50)	To	-	To	-	125150	1705.93	120100	322.98
14	co1(60)	To	-	To	-	To	-	208920	764.64
15	co10(10)	To	1800.00	10	0.06	11	0.02	10	0.02
16	co10(100)	To	-	100	42.17	101	7.46	100	0.60
17	co10(250)	To	-	250	1732.37	251	278.73	250	6.83
18	alpha2(100)	To	-	To	-	10203	741.98	10101	183.67
19	alpha2(150)	To	-	To	-	-	-	22651	1054.27
20	burns(5)	2353602	1046.92	2353602	1155.09	17382	5.14	36	0.05
21	burns(10)	To	-	To	-	To	1800.02	121	0.31
22	burns(40)	To	-	To	-	To	-	1681	185.75
23	burns(60)	To	-	To	-	To	-	3721	1532.97
24	dekker(10)	739021	420.96	739021	468.19	2713870	704.97	21	0.03
25	dekker(100)	To	-	To	-	To	1800.01	201	32.05
26	dekker(150)	To	-	To	-	To	-	301	288.25
27	dekker(200)	To	-	To	-	To	-	401	1269.84
28	petrson(5)	2782162	1432.44	2782162	1584.59	To	1800.05	31	0.04
29	petrson(50)	To	-	To	-	To	-	301	19.63
30	petrson(100)	To	-	To	-	To	-	601	474.40
31	petrson(120)	To	-	To	-	To	-	721	1186.56
32	szymnski(4)	396583	198.87	396583	221.96	1444246	319.78	5335	4.87
33	szymnski(5)	To	1800.00	To	1800.00	To	1800.04	19349	25.81
34	szymnski(7)	To	-	To	-	To	-	264209	659.04
35	na2(4,4)	2616	0.89	688	0.32	534	0.08	51	0.04
36	na2(6,6)	To	1800.00	711276	519.29	63491	6.50	2153	2.65
37	na2(14,7)	To	-	To	-	908984	128.72	18332	90.68

'Time' in seconds over 5 runs. To: Timeout = 1800s

TABLE III: Benchmarking (benchmarks with assert violation)

test		ODPOR		obs-ODPOR		RVF-SMC		ViEqui	
ID	benchmark	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time
38	na1(100,100)	1	0.22	1	0.05	1	0.04	1	0.22
39	na1(1000,500)	1	0.03	1	0.03	1	0.02	1	0.09
40	tas(20,50)	To	-	To	-	23	0.08	3	46.05
41	tas(30,50)	To	-	To	-	33	0.15	3	100.51
42	tas(40,50)	To	-	To	-	43	0.26	3	178.78
43	incdec(50)	To	-	To	-	To	-	3	9.36
44	incdec(100)	To	-	To	-	To	-	3	45.57
45	triangular(5)	20172	2.69	20172	3.12	26272	2.41	1576	0.85
46	triangular(7)	1695856	266.81	1695856	311.04	644193	70.10	32517	470.08
47	triangular(8)	To	-	To	-	3045756	360.65.10	To	-
48	FreeBSD-a	1	0.03	1	0.02	1	0.02	1	0.04
49	FreeBSD-r	1	0.02	1	0.03	1	0.01	1	0.03
50	NetBSD	4	0.03	4	0.02	6	0.02	5	0.05
51	Solaris	2	0.03	2	0.03	1	0.02	1	0.03

'Time' in seconds over 5 runs. To: Timeout = 1800s

A scatter plot contrasting performance of existing techniques against ViEqui is shown in Figure 8. Each point in the graph represents the time of analysis of the corresponding technique (on y-axis) against the time of analysis of ViEqui (on x-axis) on the tests from Table II. It can be observed that the points in the graph are concentrated near the origin of the x-axis and are scattered on the y-axis. This represents that the time of analysis of the other techniques is typically higher in comparison to that of ViEqui.

Similarly, Table II highlights that ViEqui significantly outperforms the other techniques (in terms of the time of analysis

[3] Benchmark names have been shortened in Tables II and III.

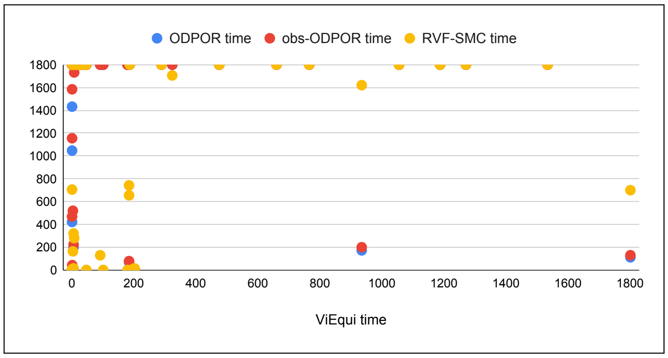


Fig. 8: Time of analysis of existing SMCs vs ViEqui (seconds)

and scalability) in providing a proof of correctness of the input program , and Table III shows that ViEqui outperforms ODPOR and obs-ODPOR and performs comparable to RVF-SMC in detecting assert violations.

The time of analysis per execution can be higher for ViEqui in comparison to the other optimal techniques, ODPOR and obs-ODPOR. The observed speedup is a result of having to consider fewer equivalence classes during examination. The said speed-up is witnessed specifically with the test IDs 4-5 (`mnbsx') and 9-11 (`rd-co'). The $|\mathcal{V}| < 4$ and the $\mathcal{E}^{\mathbb{R}}$ remains the same for both test, hence increasing the set of writes does not increase the view-equivalence classes and we witness an exponential saving in the time of analysis. In contrast, for test IDs 1-3 (`pgsql') and 6-8 (`unvrf'), the set of equivalence classes under the classical equivalence and view-equivalence is the same. We witness a slow-down on the benchmark with ViEqui, and other SMCs based on coarser equivalence relations, empirically establishing that SMCs based on coarser equivalence relations may take a higher time of analysis per execution.

obs-ODPOR shows exponential saving over ODPOR with `co10' (test IDs 15-17). The benchmark has N writes of N values and a single read at the end. However, ViEqui discovers the same set of classes exponentially faster than obs-ODPOR. `co1' and `alpha2' (test IDs 12-14,18-19) are similar benchmarks but with more reads, hence they take comparatively longer to analyze.

Benchmarks `na1' and `na2' (test IDs 35-39) concurrently update an array. The reads are performed for determining array indices, hence, the sets $\mathcal{E}^{\mathbb{R}}$ and $\mathcal{V}$ are small (for an array of length L, $|\mathcal{E}^{\mathbb{R}}| = |\mathcal{V}| = L$). Thus, ViEqui performs well on these benchmarks.

Test IDs 43-47 (`incdec', `triangular') have long causal chains of reads adding to the time of generating next-steps, however, test IDs 43-44 have, relatively, fewer reads and values allowing ViEqui to scale better. Benchmark `tas' (test IDs 40-42) showcases a scenario where forward-analysis delays the result. Various states along the execution present opportunity of forward-analysis and as a result the assert condition is reached slower.

The benchmarks of test IDs 20-34 are mutual-exclusion algorithms, that typically have large set of writes of a small set of values. ViEqui thus performs well on the tests. Test IDs 48-51 represent slices of bugs in FreeBSD, NetBSD and Solaris [21] that are successfully caught with ViEqui.

VIII. Related Work

Early efforts to tackle the combinatorial explosion of thread interleavings were static in nature with partial order reduction (POR) being a popular approach [23], [24], [25]. Seminal works like Verisoft [26], [27] and CHESS [28] popularized stateless model checking.

Stateless model checking is a popular model checking technique under sequential consistency [1], [3], [5], [6], [7], [8], [9], [10], [11] and weak memory models [2], [7], [9], [20], [29]. Several SMC techniques such as [1], [2], [5], [6], [8], [11], [29] are coupled with dynamic POR [3] to combat the state space explosion. Existing SMCs investigate various solutions to further reduce the combinatorial explosion and improve their performance, such as optimality of exploration [1], [4], [5], [9], coarser partitioning for fewer equivalence classes [5], [7], [8], [10], [11], exploration space bounding [30], [31], [32], [33] and integrating static/symbolic analysis support [34]. The techniques [4], [15] are also effective in tackling the large exploration space, however, the techniques determine equivalence by comparing program states and are essentially stateful model checking techniques.

Optimality typically comes at a cost of exponential memory use. Recent work [9] achieves optimality under reads-from equivalence with linear memory consumption.

IX. Conclusion and Future Work

This paper presents a novel view-equivalence relation for partitioning execution sequences that is at least as coarse as any existing equivalence relations. This paper also presents an SMC called ViEqui that explores an input program under view-equivalence and shows that the technique is sound, complete, and optimal. ViEqui uses a novel representation for previously explored equivalence classes called *skip* that is adequately suited for view-equivalence classes.

ViEqui technique is implemented for C/C++ input programs and is tested over 16000+ litmus tests. The paper demonstrates the effectiveness of ViEqui against existing stateless model checkers on challenging benchmarks.

Future scope. Stateless model checking under view-equivalence can be investigated for weak memory models. The applicability of view-equivalence to transactions can also be considered. ViEqui may be extended to support richer program constructs such as locks.

Appendix

Special case of $\uplus$. While computing $\overline{\mathit{Nxt}}(s_{[\tau]}) \uplus c$, consider $\exists$ $c' \in \overline{\mathit{Nxt}}(s_{[\tau]})$ and $\exists \tau_1, \tau_2$ s.t. $ns(c).\tau_1 \sim ns(c').\tau_2$, s.t.

(a) $\tau_1 = \langle\rangle$ and $ns(c') = \mathit{currNS}(s_{[\tau]})$, then replace terms of $sk(c')$ in $sk(c)$ with $\bot$ and call $\overline{\mathit{Nxt}}(s_{[\tau.ns(c')]}) \uplus (ns(c) - ns(c') \oplus \mathit{currNS}(\tau.ns(c')), \mathit{currSK}(s_{[\tau.ns(c')]}) \vee sk(c))$

(b) c' is already explored then $\forall e_r \in \mathcal{E}^{\mathbb{R}}_{ns(c')-ns(c)}$ additionally perform $\mathit{bkwdWR}(ns(c'), e_r)$ and $\mathit{bkwdRW}(ns(c'), e_r)$ while considering $\mathit{currNS}(s_{[\tau]}) = ns(c') - \langle\rangle.e_r$

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 373–384, 2014.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for tso and pso," *Acta Informatica*, vol. 54, no. 8, pp. 789–818, 2017.

[3] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *ACM Sigplan Notices*, vol. 40, no. 1, pp. 110–121, 2005.

[4] E. Albert, M. G. De La Banda, M. Gómez-Zamalloa, M. Isabel, and P. J. Stuckey, "Optimal context-sensitive dynamic partial order reduction with observers," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 352–362.

[5] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, "Optimal dynamic partial order reduction with observers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 229–248.

[6] P. A. Abdulla, M. F. Atig, B. Jonsson, M. Lång, T. P. Ngo, and K. Sagonas, "Optimal stateless model checking for reads-from equivalence under sequential consistency," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[7] T. L. Bui, K. Chatterjee, T. Gautam, A. Pavlogiannis, and V. Toman, "The reads-from equivalence for the tso and pso memory models," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[8] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, "Data-centric dynamic partial order reduction," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[9] M. Kokologiannakis, I. Marmanis, V. Gladstein, and V. Vafeiadis, "Truly stateless, optimal dynamic partial order reduction," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–28, 2022.

[10] P. Agarwal, K. Chatterjee, S. Pathak, A. Pavlogiannis, and V. Toman, "Stateless model checking under a reads-value-from equivalence," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 341–366.

[11] K. Chatterjee, A. Pavlogiannis, and V. Toman, "Value-centric dynamic partial order reduction," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[12] D. Peled, "All from one, one for all: on model checking using representatives," in *International Conference on Computer Aided Verification*. Springer, 1993, pp. 409–423.

[13] P. Godefroid, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[14] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Source sets: a foundation for optimal dynamic partial order reduction," *Journal of the ACM (JACM)*, vol. 64, no. 4, pp. 1–49, 2017.

[15] E. Albert, P. Arenas, M. G. d. l. Banda, M. Gómez-Zamalloa, and P. J. Stuckey, "Context-sensitive dynamic partial order reduction," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 526–543.

[16] Nidhugg, "Nidhugg," https://github.com/nidhugg/nidhugg, 2021.

[17] A. Mazurkiewicz, "Trace theory," in *Advanced course on Petri nets*. Springer, 1986, pp. 278–324.

[18] H. T. Nguyen, C. Rodríguez, M. Sousa, C. Coti, and L. Petrucci, "Quasi-optimal partial order reduction," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 354–371.

[19] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," in *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation*, 2015, pp. 250–259.

[20] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, "Optimal stateless model checking under the release-acquire semantics," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[21] D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 401–422.

[22] Mc-imperial, "Sctbench," https://github.com/mc-imperial/sctbench, 2022.

[23] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[24] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, vol. 2, no. 2, pp. 149–164, 1993.

[25] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Static partial order reduction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 345–357.

[26] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 174–186.

[27] P. Godfroid, "Software model checking: The verisoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[28] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, no. 2008, 2008.

[29] S. Singh, D. Sharma, and S. Sharma, "Dynamic verification of c11 concurrency over multi copy atomics," in *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2021, pp. 39–46.

[30] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of c and verilog programs using bounded model checking," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE, 2003, pp. 368–371.

[31] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.

[32] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[33] A. Udupa, A. Desai, and S. Rajamani, "Depth bounded explicit-state model checking," in *International SPIN Workshop on Model Checking of Software*. Springer, 2011, pp. 57–74.

[34] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, "Unfolding-based partial order reduction," in *International Conference on Concurrency Theory*, 2015, p. 456–469.