# Extracting Safe Thread Schedules from Incomplete Model Checking Results

Patrick Metzler[1], Neeraj Suri[1], and Georg Weissenbacher[2]

[1] Technische Universität Darmstadt
[2] TU Wien

**Abstract.** Model checkers frequently fail to completely verify a concurrent program, even if partial-order reduction is applied. The verification engineer is left in doubt whether the program is safe and the effort towards verifying the program is wasted.
We present a technique that uses the results of such incomplete verification attempts to construct a (fair) scheduler that allows the safe execution of the partially verified concurrent program. This scheduler restricts the execution to schedules that have been proven safe (and prevents executions that were found to be erroneous). We evaluate the performance of our technique and show how it can be improved using partial-order reduction. While constraining the scheduler results in a considerable performance penalty in general, we show that in some cases our approach—somewhat surprisingly—even leads to faster executions.

## 1 Introduction

Automated verification of concurrent programs is inherently difficult because of exponentially large state spaces [39]. State space reductions such as partial-order reduction (POR) [10,17,16] allow a model checker to focus on a subset of all reachable states while the verification result is valid for all reachable states. However, even reduced state spaces may be intractably large [17] and corresponding programs infeasible to (automatically) verify, requiring manual intervention.

We propose a novel model checking approach for safety verification of potentially non-terminating programs with a bounded number of threads, nondeterministic scheduling, and shared memory. Our approach iteratively generates *incomplete verification results* (IVRs) to prove the safety of a program under a (semi-)deterministic scheduler. By enforcing the scheduling constraints induced by an IVR (cf. *iteratively relaxed scheduling* [30]), all executions (under all nondeterministic inputs) are safe, even if the underlying (operating system) scheduler is non-deterministic. Thereby, the program can be executed safely before a (potentially infeasible) complete verification result is available. Executions can still exploit concurrency and the number of memory accesses that are executed concurrently may even be increased. As the model checking problem is eased, additional programs become tractable. Furthermore, IVRs can be used to safely execute unsafe programs which are safe under at least one scheduler. E.g., instead of programming synchronization explicitly, our model checking algorithm can be used to synthesize synchronization so that all executions are safe.

We use the producer-consumer example from Fig. 1 to explain our approach. The verifier analyses an initial schedule, e.g., where thread $T_1$ and $T_2$ produce and consume in turns, and emits an IVR $\mathcal{R}_1$, guaranteeing safe executions under this schedule. With its second IVR, the verifier might verify the correctness of producing two items in a row and the scheduling constraints can be relaxed accordingly. When the verifier hits an unsafe execution (the consumer produces an underflow), it emits an unsafe IVR for debugging. If the verifier accomplishes to analyze all possible executions of the program, it will report the final result *partially safe*, as the program can be used safely under all inputs but unsafe executions exist. Had there been no unsafe or safe IVR, the final result would be *safe* or *unsafe*, respectively.

```
1  initially:
2    empty buffer of
           size N
3    count = 0
4    mutex = 0
5  thread T1:
6    while true:
7      produce()
8  thread T2:
9    while true:
10     consume()
11 produce:
12   lock(mutex)
13   if count < N:
14     put item
15     count += 1
16   unlock(mutex)
17 consume:
18   lock(mutex)
19   remove item
20   count −= 1
21   unlock(mutex)
```

Fig. 1: Producer-consumer problem with bug

This paper shows how to instantiate our approach by answering these questions: 1. Which state space abstractions are suitable for iterative model checking? The abstraction should be able to represent non-terminating executions and facilitate the extraction of schedules. 2. How to formalize and represent suitable IVRs? IVRs should be as small as possible in order to allow short iterations, while they must be large enough to guarantee fully functional executions under all possible program inputs. More precisely, for every possible program input, an IVR must cover a program execution. 3. What are suitable model checking algorithms that can be adapted to produce IVRs? A suitable algorithm should easily allow to select schedules for exploration.
*We provide an extended presentation of our formal framework with proofs in Appendices A–C, as well as additional experimental results in Appendix D.*

## 2 Incomplete verification results

### 2.1 Basic definitions

A *program* $P$ comprises a set $S$ of states (including a distinct initial state) and a finite set $\mathscr{T}$ of threads. Each state $s \in S$ maps program counters and variables to values. We use $\mathfrak{l}(s)$ to denote the program location of a state $s$, which comprises a local location $\mathfrak{l}_T(s)$ for each thread $T \in \mathscr{T}$. W.l.o.g. we assume the existence of a single error location that is only reachable if the program $P$ is not safe.

A state formula $\phi$ is a predicate over the program variables encoding all states $s$ in which $\phi(s)$ evaluates to true. A transition relation $R$ relates states $s$ and their successor states $s'$. Each tread $T$ is partitioned into local transitions $R_{\mathfrak{l},\mathfrak{l}'}$ such that $\mathfrak{l} = \mathfrak{l}_T(s)$ and $\mathfrak{l}' = \mathfrak{l}_T(s')$ for all $s, s'$ satisfying $R_{\mathfrak{l},\mathfrak{l}'}(s, s')$ and $R_{\mathfrak{l},\mathfrak{l}'}$ leaves the program locations and variables of other threads unchanged. We use $\mathit{Guard}(R)$ to denote a predicate encoding $\exists s' \,.\, R(s, s')$, e.g., $\mathit{Guard}(R_{13,14})$ is $(\mathsf{count} < \mathsf{N})$ for the transition from location 13 to 14 in Fig. 1. We say that $R_{\mathfrak{l},\mathfrak{l}'}$ (or $T$, respectively) is *active* at location $\mathfrak{l}$ and *enabled* in a state $s$ iff $\mathfrak{l}(s) = \mathfrak{l}$

and $s$ satisifes $Guard(R)$. Multiple transitions of a thread $T$ at a location can be active, but we allow only one transition $R$ to be enabled at a given state and define $enabled_T(s) := \{R\}$ if $R$ exists and $enabled_T(s) := \emptyset$ otherwise.

If there exist states $s$ for which no transition of a thread $T$ is enabled (e.g., in line 12 in Fig. 1), $T$ may block. We assume that such locations $\mathfrak{l}_T(s)$ are (conservatively) marked by $may\text{-}block(\mathfrak{l}_T(s))$.

An *execution* is a sequence $s_0, T_1, s_1, \ldots$, where $s_0$ is the initial state and the states $s_i$ and $s_{i+1}$ in every adjacent triple $(s_i, T_i, s_{i+1})$ are related by the transition relation of $T_i$. An execution that does not reach the error location is *safe*. A *deadlock* is a state $s$ in which no transitions are enabled. W.l.o.g. we assume that all finite executions correspond to deadlocks and are undesirable; intentionally terminating executions can be modelled using terminal locations with self-loops.

An execution $\tau$ is (strongly) *fair* if every thread $T_i$ enabled infinitely often in $\tau$ is also scheduled infinitely often [5]. We assume that fairness is desirable.

Non-determinism can arise both through scheduling and non-deterministic transitions. A *scheduler* can resolve the former kind of non-determinism.

**Definition 1 (scheduler).** *A* scheduler $\zeta : (S \times \mathscr{T})^* \times S \to \mathscr{T}$ *of a program $P$ is a function that takes an execution prefix $s_0, T_1, \ldots, T_n, s_n$ and selects a thread that is enabled at $s_n$, if such a thread exists. A scheduler $\zeta$ is* deadlock-free *(*fair*, respectively) if all executions possible under $\zeta$ are deadlock-free (fair).*

A scheduler for the program of Fig. 1, for instance, must select $T_1$ rather than $T_2$ for the prefix $s_{init}, T_1, s_1, T_1, s_2, T_1, s_3, T_2, s_4, T_2, s_5$, since at that point the lock is held by $T_1$ and $enabled_{T_2}(s_5) = \emptyset$.

Non-deterministic transitions are the second source of non-determinism. If $R_{\mathfrak{l},\mathfrak{l}'}$ of thread $T$ allows multiple successor states for a state $s$, we presume the existence of input symbols $X$ such that each $\iota \in X$ determines a unique successor state $s'$ by selecting an $R^{\iota}_{\mathfrak{l},\mathfrak{l}'} \subseteq R_{\mathfrak{l},\mathfrak{l}'}$ with $R^{\iota}_{\mathfrak{l},\mathfrak{l}'}(s, s')$.

**Definition 2 (input).** *An* input *is a function $\chi : (S \times \mathscr{T})^* \to X$, which chooses an input symbol depending on the current execution prefix.*

In conjunction, an input and a scheduler render a program completely deterministic: the input $\chi$ and scheduler $\zeta$ select a transition in each step such that each adjacent triple $(s_i, T_{i+1}, s_{i+1})$ is uniquely determined.

For Partial Order Reduction (POR), we assume that a symmetric independence relation $\parallel$ on transitions of different threads is given, which induces an equivalence relation on executions. Two transitions $R_1$ and $R_2$ are only independent if they are from distinct threads, they are commutative at states where both $R_1$ and $R_2$ are enabled, and executing $R_1$ does neither enable nor disable $R_2$. We write $R_1 \nparallel R_2$ if $R_1$ and $R_2$ are not independent.

### 2.2 Requirements on incomplete verification results

Our goal is to ease the verification task by producing incomplete verification results (IVRs) which prove the program safety under reduced non-determinism,

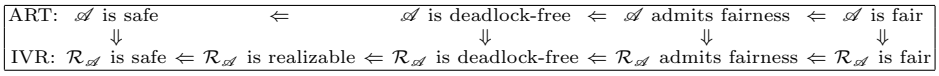



Fig. 2: Overview on the relationship between properties of IVRs and ARTs

i.e., only for a certain scheduler. We only allow "legitimate" restrictions of the scheduler that do not introduce deadlocks or exclude threads. Inputs must not be restricted, since this might reduce functionality and result in unhandled inputs.

Hence, we define an IVR to be a function $\mathcal{R}$ that maps execution prefixes to sets of threads, representing scheduling constraints. An IVR for the program from Fig. 1, for instance, may output $\{T_1\}$ in states with an empty buffer, meaning that only thread $T_1$ may be scheduled here, and $\{T_2\}$ otherwise, so that an item is produced if and only if the buffer is empty. A scheduler *enforces* (the scheduling constraints of) an IVR $\mathcal{R}$ if $\zeta_{\mathcal{R}}(\tau) \in \mathcal{R}(\tau)$ for all execution prefixes $\tau$. IVR $\mathcal{R}$ *permits* all executions possible under a scheduler that enforces $\mathcal{R}$.

The remainder of this subsection discusses the requirements on useful IVRs. We define *safe*, *realizable*, *deadlock-free*, *fairness-admitting*, and *fair* IVRs. In the following subsection, we instantiate IVRs with abstract reachability trees (ARTs). Fig. 2 gives an overview on the logical relationship between properties of IVRs and ARTs.

*Safety.* An IVR $\mathcal{R}$ can either expose a bug in a program or guarantee that all permitted executions are safe. Here, we are only concerned with the latter case. An IVR $\mathcal{R}$ is *safe* if all executions permitted by $\mathcal{R}$ are safe. An unsafe IVR permits an unsafe execution and is called a *counterexample*.

*Completeness.* To reduce the work for the model checker, a safe IVR $\mathcal{R}$ should ideally have to prove the correctness of as few executions as possible. At the same time, it should cover sufficiently many executions so that the program can be used without functional restrictions. For instance, the IVR $\mathcal{R}(\tau) := \emptyset$, for all $\tau$, is safe but not useful, as it does not permit any execution. Consequently, $\mathcal{R}$ should permit at least one enabled transition, in all non-deadlock states, which is done by *realizable* IVRs: an IVR $\mathcal{R}$ is *realizable* if at least one scheduler that enforces $\mathcal{R}$ exists. Furthermore, an IVR should never introduce a deadlock: an IVR $\mathcal{R}$ is *deadlock-free* if all schedulers that enforce $\mathcal{R}$ are deadlock-free.

*Fairness.* In general, we deem only fair executions desirable. The IVR $\mathcal{R}(\tau) := \{T_1\}$, for instance, is deadlock-free for the program of Fig. 1 but useless, as no item is consumed. A deadlock-free IVR *admits fairness* if there exists a fair scheduler enforcing $\mathcal{R}$ (i.e., a fair execution of the program is possible).

If a scheduler permits both fair and unfair executions, it might be difficult to guarantee fairness at runtime. In such cases, a *fair* IVR can be used: A deadlock-free IVR $\mathcal{R}$ is *fair* if all schedulers enforcing $\mathcal{R}$ are fair.

### 2.3 Abstract reachability trees as incomplete verification results

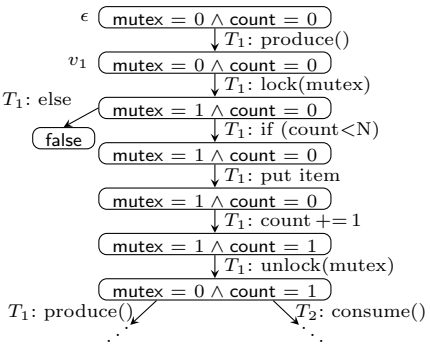
In this subsection, we instantiate the notion of IVRs using abstract reachability trees (ARTs), which underly a range of software model checking tools [21,29,24,9] and have recently been used for concurrent programs [40]. Due to the explicit

representation of scheduling choices from the beginning of an execution up to an (abstract) state, ARTs are well-suited to represent IVRs. Model checking algorithms based on ARTs perform a path-wise exploration of program executions and represent the current state of the exploration using a tree in which each node $v$ corresponds to a set of states at a program location $\mathfrak{l}(v)$. These states, represented by a predicate $\phi(v)$, (safely) over-approximate the states reachable via the program path from the root of the ART ($\epsilon$) to $v$. Edges expanded at $v$ correspond to transitions starting at $\mathfrak{l}(v)$. A node $w$ may *cover* $v$ (written $v \rhd w$) if the states at $w$ include all states at $v$ ($\phi(v) \Rightarrow \phi(w)$); in this cases, $v$ is covered (*covered*$(v)$) and its successors need not be further explored. (Intuitively, executions reaching $v$ are continued from $w$.) Formally, an ART is defined as follows:

**Definition 3 (abstract reachability tree [29,40]).** *An* abstract reachability tree *(ART) is a tuple $\mathscr{A} = (V, \epsilon, \rightarrow, \rhd)$, where $(V, \rightarrow)$ is a finite tree with root $\epsilon \in V$ and $\rhd \subseteq V \times V$ is a covering relation. Nodes $v$ are labeled with global control locations and state formulas, written $\mathfrak{l}(v)$ and $\phi(v)$, respectively. Edges $(v, w) \in \rightarrow$ are labeled with a thread and a transition, written $v \xrightarrow{T,R} w$.*

ART $\mathscr{A}$ is *well-labeled* [29] if for every edge $v \xrightarrow{T,R_{\mathfrak{l},\mathfrak{l}'}} w$ in $\mathscr{A}$ we have that (i) $\phi(\epsilon)$ represents initial state, (ii) $\phi(v)(s) \wedge R_{\mathfrak{l},\mathfrak{l}'}(s, s') \Rightarrow \phi(w)(s')$ and $\mathfrak{l}_T(v) = \mathfrak{l}$ and $\mathfrak{l}_T(w) = \mathfrak{l}'$, and (iii) for every $v, w$ with $v \rhd w$, $\phi(v) \Rightarrow \phi(w)$ and $\neg$*covered*$(w)$.

An incomplete ART $\mathscr{A}_{p-c}$ for the producer-consumer problem of Fig. 1 is shown on the right. Nodes show the state formulas and edges are labeled with the thread and statement corresponding to the transition.




*ART-induced schedulers.* A well-labeled ART $\mathscr{A}$ directly corresponds to an IVR $\mathcal{R}_{\mathscr{A}}$ that simulates an execution by traversing $\mathscr{A}$. We define $\mathcal{R}_{\mathscr{A}}$ as follows: Let $\tau = s_0, T_1, s_1, \ldots, s_n$ be an execution prefix. If $\mathscr{A}$ contains no path that corresponds to $\tau$, $\mathcal{R}_{\mathscr{A}}$ leaves the schedules for this execution unconstrained. Otherwise, let $v_n$ be the last node of the path in $\mathscr{A}$ that corresponds to $\tau$. $\mathcal{R}_{\mathscr{A}}$ permits exactly those threads that are expanded at $v_n$ (or at $w$ if $v_n$ is covered by some node $w$). E.g., the execution prefix $\tau = s_0, T_1, s_1$ corresponds to the path from $\epsilon$ to $v_1$ in $\mathscr{A}_{p-c}$. As only $T_1$ is expanded at $v_1$, $\mathcal{R}_{\mathscr{A}\,p-c}$ allows only $\{T_1\}$ after $\tau$.

*Safety.* An ART is *safe* if whenever $\mathfrak{l}_T(v)$ is the error location then $\phi(v) =$ *false*. As only safe executions may correspond to a path in a safe ART (cf. Theorem 3.3 of [40]), $\mathcal{R}_{\mathscr{A}}$ is a safe IVR.

*Completeness.* In order to derive a deadlock-free IVR from a well-labeled ART $\mathscr{A}$, we have to fully expand at least one thread $T$ at each node $v$ that represents reachable states (where $T$ is fully expanded at $v$ if $v$ has an outgoing edge for every active transition of $T$ at $\mathfrak{l}_T(v)$). However, there may exist reachable states

$s$ represented by $\phi(v)$ for which no action of $T$ is enabled (i.e., $enabled_T(s) = \emptyset$). If $T$ is the only thread expanded at $v$, $\mathcal{R}_{\mathscr{A}}$ is not realizable. This situation can arise for locations $\mathfrak{l}$ at which $T$ may block (marked with $may\text{-}block(\mathfrak{l}_T)$).

Consequently, whenever $may\text{-}block(\mathfrak{l}_T(v))$ in a *deadlock-free* ART $\mathscr{A}$, we require that $\phi(v)$ is strong enough to entail that the transitions $R$ of $T$ expanded at $v$ (or at the node covering $v$, respectively) are enabled (i.e., $\phi(v) \Rightarrow Guard(R)$). For instance, $\phi(v_1)$ in the ART shown above proves the enabledness of $T_1$ at $v_1$, as $\phi(v_1) \Rightarrow \text{mutex} = 0$ and lock(mutex) is enabled if mutex $= 0$.

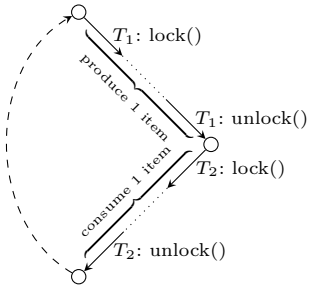
**Lemma 1.** *If an ART $\mathscr{A}$ is deadlock-free, $\mathcal{R}_{\mathscr{A}}$ is a deadlock-free IVR.*

*Fairness.* IVRs derived from deadlock-free ARTs do not necessarily admit fairness if the underlying ART contains cycles (across $\triangleright$ and $\rightarrow$ edges) that represent unfair executions. In order to make sure a deadlock-free ART *admits fairness* we implement a scheduler that allows $\mathscr{A}$ to schedule each thread infinitely often (whenever it is enabled infinitely often) by requiring that every $(\triangleright \cup \rightarrow)$-cycle is "fair", defined as follows.

**Definition 4 (ART admitting fairness).** *A deadlock-free ART $\mathscr{A} = (V, \epsilon, \rightarrow, \triangleright)$* admits fairness *if every $(\triangleright \cup \rightarrow)$-cycle contains, for every thread $T$ that is enabled at a node of the cycle, a node $v$ such that $T$ is expanded at $v$.*

**Lemma 2.** *If an ART $\mathscr{A}$ admits fairness, $\mathcal{R}_{\mathscr{A}}$ is an IVR that admits fairness.*

Note that the expansion of a thread $T$ at a node in a cycle does not guarantee that the transition is part of the cycle. A slight modification of the fairness condition for ARTs leads to a sufficient condition for ARTs as fair IVRs, as the following definition and lemma show. The difference in the fairness condition is that all enabled threads are expanded *within* each $(\triangleright \cup \rightarrow)$-cycle $c$, which we denote by $fair(c)$. The $(\triangleright \cup \rightarrow)$-cycle shown on the right, for instance, is fair.




**Definition 5 (fair ART).** *A deadlock-free ART $\mathscr{A} = (V, \epsilon, \rightarrow, \triangleright)$ is* fair *if $fair(c)$ holds for every $(\triangleright \cup \rightarrow)$-cycle $c$.*

**Lemma 3 (fairness).** *For all fair ARTs $\mathscr{A}$, $\mathcal{R}_{\mathscr{A}}$ is a fair IVR.*

Given an ART $\mathscr{A}$ that admits fairness, one can generate a fair ART $\mathscr{A}'$ such that $\mathcal{R}_{\mathscr{A}}$ permits all executions permitted by $\mathcal{R}_{\mathscr{A}'}$, as shown in Appendix G.

## 3 Iterative model checking

A suitable algorithm for our framework must generate fair IVRs. We use model checking based on ARTs (cf. Sec. 2.3), which allows us to check infinite executions and explicitly represent scheduling. Nevertheless, other program analysis

**Algorithm 1:** Iterative IMPACT for concurrent programs: main procedure (based on [40]) — part I

**input** : Program with threads $\mathscr{T}$
**intermediate outputs:** fair ARTs $\mathscr{A}_1 \subseteq \mathscr{A}_2 \subseteq \ldots \subseteq \mathscr{A}_n$ and unsafe ARTs
**output** : safe, partially safe, or unsafe

**Data:** $\mathscr{A} = (V, \epsilon, \rightarrow, \rhd) := (\{\epsilon\}, \epsilon, \emptyset, \emptyset)$, $W := \{\epsilon\}$, $I := \{\}$

```
1  Function Main()
2      while true do
3          status := Iteration()
4          if status = no progress then
5              break
6          else if status = counterexample
                   then
7              yield 𝒜 as an unsafe IVR
8          else
9              𝒜′ := Remove_Error_Paths(𝒜)
10             yield 𝒜′ as a safe IVR
11     if 𝒜 is safe then
12         return safe
13     else if Remove_Error_Paths(𝒜)
                admits fairness then
14         return partially-safe
15     else
16         return unsafe
17 Function Iteration()
18     W := New_Schedule_Start()
19     if W = ∅ then
20         return no progress
21     while W ≠ ∅ do
22         select and remove v from W
23         Close(v)
24         if v not covered then
25             status := Refine (v)
26             if status = counterexample then
27                 return counterexample
28             status := Check_Enabledness(v)
29             if status = no progress then
30                 return no progress
31             Expand (v)
32     return progress
```

techniques such as symbolic execution are also suitable to generate IVRs. In particular, our algorithm (Alg. 1 and 2) constitutes an iterative extension of the IMPACT algorithm [29] for concurrent programs [40]. We chose IMPACT as a base for our algorithm because it has an available implementation for multi-threaded programs, which we use to evaluate our approach in Sec. 5.

IMPACT generates an ART by path-wise unwinding the transitions of a program. Once an error location is reached at a node $v$, IMPACT checks whether the path $\pi$ from the ART's root to $v$ corresponds to a feasible execution. If this is the case, a property violation is reported; otherwise, a safety invariant for $\pi$ is generated via interpolation and the node labeling is updated. Thereby, a well-labeled ART is maintained. Once the ART is complete, its node labeling provides a safety proof for the program.

In each iteration, our extended algorithm yields an IVR which is either unsafe (a counterexample) or fair (can be used as scheduling constraints). If the algorithm terminates, it outputs "safe", "partially safe", or "unsafe", depending on whether the program is safe under all, some, or no schedulers. Procedure *Main()* repeatedly calls *Iteration()* (line 3), which, intuitively, corresponds to an execution of the original algorithm of [40] under a deterministic scheduler. *Iteration()* (potentially) extends the ART $\mathscr{A}$. If no progress is made ($\mathscr{A}$ is unchanged), the algorithm terminates and reports "safe", "partially safe", or "unsafe"(lines 12, 14, and 16). If *Iteration()* produces a counterexample $\mathscr{A}$, the ART is yielded as an intermediate output (line 7). Otherwise, *Iteration()* has found a new IVR, which is yielded as an intermediate output (line 10). This IVR corresponds to $\mathscr{A}$ with all previously found counterexamples removed, i.e., the largest fair ART that is a subgraph of $\mathscr{A}$, denoted by *Remove_Error_Paths()*.

**Algorithm 2:** Iterative IMPACT for concurrent programs — part II

```
   continued:
 1 Function Check_Enabledness(v)
 2     π := v_0 --T_1,R_1--> v_1 ... --T_n,R_n--> v_n
             path from ε to v
 3     if not may-block(ł(v_{n-1}) T_n then
 4         return progress
 5     if R_1 ∧ ... ∧ R_{n-1} ∧ ¬Guard(R_n) is
             unsat then
 6         φ(v) := φ(v) ∧ Guard(R_n)
 7     else
 8         return Backtrack(v)
 9 Function Close(v)
10     for all uncovered nodes w that have
             been created before v do
11         if ł(w) = ł(v) ∧ (φ(v) ⇒ φ(w))
                 ∧∀c ∈ C_A(v, w). fair(c) then
12             ▷ :=▷ ∪{(v, w)}
13             ▷ :=▷ \{(x, y) : v ⇝ y}
14             for T with v --T--> v'
                       and not w --T--> w' do
15                 add (v, T) to I
16 Function Backtrack(v)
17     π := v_0 --T_1,R_1--> v_1 ... --T_n,R_n--> v_n
             path from ε to v
18     i := n − 1
19     while i ≥ 0 do
20         if ∃T, v'_i. v_i --T--> v'_i ∉ A
                 ∧(Skip(v_i, T) = false) then
21             add v_i --T--> v'_i to A
22             W := W ∪ {v'_i}
23             prune --T_{i+2},R_{i+2}--> v_{i+3} ...
                       ... --T_n,R_n--> v_n from A
24             φ(v_{i+1}) := false
25             return progress
26         i := i − 1
27     return no progress
28
29 Function Expand(v)
30     T := Schedule_Thread (v)
31     Expand_Thread (T, v)
```

*Iteration()* maintains a work list $W$ of nodes $v$ to be explored via *Close(v)* (Alg. 2), which tries to find (as in [40]) a node that covers $v$. In addition to the covering check of [40], we check fairness, i.e., a covering is only added if no unfair cycle arises, where $C_{\mathscr{A}}(v, w)$ denotes all cycles that would be closed by adding the edge $v \rhd w$ (line 11 of Alg. 2). If such a node $w$ is found, any thread $T$ that is expanded at $v$ but not at $w$ (line 14 of Alg. 2) must not be skipped at $w$ by POR. Instead of expanding $T$ instantaneously at $w$ (as in [40]), which would result in the exploration of two schedules in the same iteration, $T$ is added to the set $I$ so that it can be explored in a subsequent iteration (for a different schedule). If no covering node for $v$ is found, the same refinement procedure as in [40], extended with a return value *counterexample* representing feasible error paths, is called (line 25). If the path to $v$ is not a feasible error path (line 28 of Alg. 1), *Check_Enabledness()* (Alg. 2) performs a deadlock check by testing whether the last action that leads to $v$ is enabled in all states represented by the predecessor node. If not, deadlock-freedom is not guaranteed and *Backtrack()* tries to find a substitute node where exploration can continue.

The deterministic scheduler of *Iteration()* is controlled by *New_Schedule_Start()* and *Schedule_Thread()*. The former selects a set of initial nodes for the exploration (line 18 of Alg. 1); the latter decides which thread to expand at a given node (line 30 of Alg. 2). We use a simple heuristic that selects the first (in breadth-first order) node which is not yet fully expanded and use a round-robin scheduler for *Schedule_Thread* that switches to the next thread once a back jump occurs (e.g., the end of a loop body is reached). Additionally, *Schedule_Thread* returns only threads that are necessary to expand at the given node after POR (cf. *Skip()* [40]). More elaborate heuristics are conceivable but out of the scope of this paper. An extended presentation of our algorithm is provided in Appendix H.

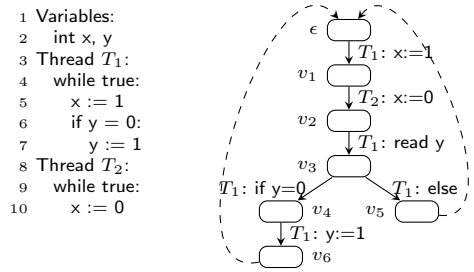



Fig. 3 (a) Section paths

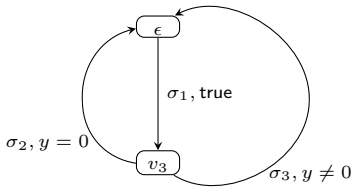



Fig. 3 (b) A program schedule

The correctness of Alg. 1 w.r.t. safety follows from the correctness of [29] and [40]. Additionally, Alg. 1 is also fair (a proof is provided in Appendix C):

**Lemma 4 (fairness of Alg. 1).** *Any safe ART $\mathscr{A}$ generated by Alg. 1 is fair.*

## 4 Partial-order reduction

A naive enforcement of the context switches at the relevant nodes of a safe IVR $\mathcal{R}_{\mathscr{A}}$ would result in a strictly sequential execution of the transitions, foiling any benefits of concurrency. To enable parallel executions, we introduce *program schedules* that relax the scheduling constraints by means of partial-order reduction (POR). (More details including proofs are given in Appendix B.) Note that this application of POR concerns the enforcement of scheduling constraints and occurs in addition to POR applied by our model checking algorithm when constructing an ART (cf. Sec. 3). Nevertheless, dependency information that is used for POR during model checking can be reused so that redundant computations are avoided.

The goal is to permit the parallel execution of independent transitions (in different threads) whose order does not affect the outcome of the execution represented by $\mathscr{A}$ (i.e., the resulting traces are Mazurkiewicz-equivalent). Using traditional POR to construct such scheduling constraints poses two challenges: 1. Executions may be infinite, but we need a finite representation of scheduling constraints. 2. The control flow of an execution may be unpredictable, i.e., it is a priori unclear which scheduling constraints will apply. We solve issue 1 by partitioning ARTs into *sections* and associate a finite schedule with every section. To address issue 2, we require that sections do not contain branchings (control flow and non-deterministic transitions).

Consider the program and corresponding ART in Fig. 3a. The if-statement of $T_1$ is modeled as a separate read transition followed by a branching at node $v_3$. We define three section paths $\pi_1 := \epsilon \to v_1 \to v_2 \to v_3$, $\pi_2 := v_3 \to v_4 \to v_6 \to \epsilon$, and $\pi_3 := v_3 \to v_5 \to \epsilon$. After $\pi_1$ has been executed, a scheduler can distinguish the cases $y = 0$ and $y \neq 0$ and schedule $\pi_2$ or $\pi_3$ accordingly.

Formally, a *section path* $v_1 \xrightarrow{R_1} \ldots \xrightarrow{R_n} v_{n+1}$ corresponds to a branching-free path in an ART whose first transition may be guarded. A section path follows $\rightarrow_{\mathscr{A}}$ edges, skipping covering edges $\triangleright$. The *section schedule* of a section path describes the Mazurkiewicz equivalence class of the contained transitions and is defined as the smallest partial order $\sigma = (V_\sigma, \rightarrow_\sigma)$ such that $V_\sigma = \{e_1, \ldots, e_n\}$ and $\rightarrow_\sigma \supseteq \{(e_i, e_j) : i < j \wedge R_i \not\parallel R_j\}$, where $e_i, 1 \leq i \leq n$ is the occurrence of transition $R_i$ at position $i$. The section schedule of $\pi_1$ is $(\{e_1, e_2, e_3\}, \{(e_1, e_2), (e_1, e_3)\})$ with $e_1 \triangleq T_1 : \texttt{x:=1}$, $e_2 \triangleq T_2 : \texttt{x:=0}$, and $e_3 \triangleq T_1 : \textsf{read y}$.

A *program schedule* $\Sigma$ comprises several section schedules. $\Sigma$ is a labeled graph $(V_\Sigma, \rightarrow_\Sigma)$. Each node $v \in V_\Sigma$ is the start of a section path $\pi$ in $\mathscr{A}$. Each edge is labeled with the section schedule of $\pi$ and the guard $\mathit{Guard}(R)$ of the first transition $R$ in $\pi$. As $\mathscr{A}$ is deadlock-free, there exists a thread $T$ which is fully expanded at $v$ in $\mathscr{A}$ and we require that $\Sigma$ likewise has outgoing edges at $v$ labeled with $T$ for each transition of $T$ at $v$. Fig. 3b shows a program schedule for our example program.

A scheduler can enforce the scheduling constraints of a program schedule by picking a section schedule that matches the current execution prefix and scheduling an event whose predecessors (according to the section schedule) have already been executed. Hence, all independent events in a section can be executed concurrently without synchronization. All events of a section schedule have to appear before the first event of the next section schedule, so that the states reached between sections correspond to nodes of the program schedule.

A program schedule of an ART $\mathscr{A}$ that admits fairness permits exactly those executions that correspond to a path in $\mathscr{A}$ (modulo Mazurkiewicz equivalence). In particular, as Mazurkiewicz equivalence preserves safety properties [17], only safe executions are permitted.

**Lemma 5 (correctness).** *Let $\mathscr{A}$ be an ART that admits fairness and $\Sigma$ a program schedule for $\mathscr{A}$. All program executions induced by $\Sigma$ are equivalent to an execution that corresponds to a path in $\mathscr{A}$.*

## 5 Evaluation

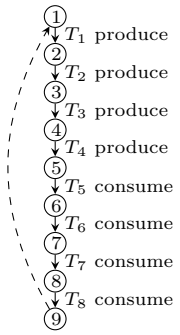
In five case studies, we evaluate our iterative model checking algorithm and scheduling based on IVRs. We use the Impara model checker [40], as it is the only available implementation of model checking for non-terminating, multi-threaded programs based on a forward analysis on ARTs we have found. Impara uses lazy abstraction with interpolants based on weakest preconditions. We extend the tool by implementing our algorithm presented in Sec. 3. Impara accepts C programs as inputs, however, some language features are not supported and we have rewritten programs accordingly.[3] We refer to the (non-iterative) Impara

[3] E.g., Pthread mutexes, some uses of the address-of operator, and reuse of the same function by several threads are not supported. We solve these issues by rewriting our benchmark programs so that Impara handles them correctly and their intuitive semantics is not changed. We will publish our modifications to Impara, including two bug fixes.

tool as Impara-C (for complete verification) and to our extension of Impara with iterative model checking as Impara-IMC.

Based on the ARTs constructed by Impara, program schedules are generated automatically and encoded as vector clocks. We instrument the benchmark programs with a call-back to a specially designed user space scheduler directly before and after each access to a global variable. The result is a multi-threaded program that executes concurrent memory accesses according to a given program schedule. All experiments have been executed on a 4-core Intel Core i5-6500 CPU at 3.2 GHz. We report median values averaged over five runs.

### 5.1 Infeasible complete verification




Even for a moderate number of threads, complete verification, i.e., verification of a program under all possible schedules and inputs, may be infeasible. In particular, Impara-C times out (after 72 h) on a corrected variant of the producer consumer problem (Fig. 1) with four producers and four consumers. Impara-IMC produces the first IVR $\mathcal{R}_1$ after 4:29:53 hours. A simplification of $\mathcal{R}_1$ is depicted on the right; it covers all executions in which the threads appear to execute their loop bodies atomically in the order $T_1, T_2, \ldots, T_8$.

The subsequent IVRs $\mathcal{R}_2, \ldots, \mathcal{R}_8$ are found much faster than the first IVR, after 19:31, 12:3, 6:13, 28:0, 9:25, 8:27, and 8:40 minutes. We stop the model checker after eight IVRs. According to our implementation of *New_Schedule_Start()* in Alg. 1, IVR $\mathcal{R}_i$ permits, in addition to all executions permitted by $\mathcal{R}_{i-1}$, those executions in which the threads appear in the order $T_i, T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_8$. Hence, $\mathcal{R}_8$ gives the scheduler more freedom than $\mathcal{R}_1$, which may result in a better execution performance, e.g., because a producer which has its item available earlier does not have to wait for all previous producers.

### 5.2 Deadlocks

```
1  Thread T1:
2    while true:
3      lock(mutex1)
4      lock(mutex2)
5      execute_critical_section()
6      unlock(mutex2)
7      unlock(mutex1)
```

```
8  Thread T2:
9    while true:
10     lock(mutex2)
11     lock(mutex1)
12     execute_critical_section()
13     unlock(mutex2)
14     unlock(mutex1)
```

A common issue with multi-threaded programs are deadlocks, which may occur when multiple mutexes are acquired in a wrong order, as in the program on the right, in which two threads use two mutexes to protect their critical sections. A deadlock is reached, e.g., when $T_2$ acquires mutex2 directly after $T_1$ has acquired mutex1. A monolithic verification approach would try to verify one or more executions and, as soon as a deadlock is found, report the execution that leads to the deadlock as a counterexample. With manual intervention, this counterexample can be inspected in order to identify and fix the bug.

In contrast, Impara-IMC logs both safe and unsafe IVRs. The first IVR found in this example covers all executions in which Threads 1 and 2 execute

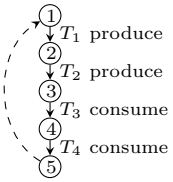
their loop bodies in turns, with Thread 1 beginning. As expected, executing the program with enforcing the first program schedule never leads to a deadlock. Executing the uninstrumented program (without scheduling constraints) leads to a deadlock after only a few hundred loop iterations. Hence, IMC enables to safely use the program deadlock-free and without manual intervention.

### 5.3 Race conditions through erroneous synchronization

```
1  Threads
2    T1: while true: produce()
3    T2: while true: produce()
4    T3: while true: consume()
5    T4: while true: consume()
6  produce:
7    if buffer_is_not_full():
8      lock()
9      assert buffer_is_not_full()
10     add_item()
11     unlock()
12 consume:
13   if buffer_is_not_empty():
14     lock()
15     assert buffer_is_not_empty()
16     remove_item()
17     unlock()
```

The above program shows a variant of the producer-consumer problem with two producers and two consumers which uses erroneous synchronization: both the produce and consume check the amount of free space without acquiring the mutex first. For example, a buffer underflow occurs if the buffer contains only one item and the two consumers concurrently find that the buffer is not empty; although the buffer becomes empty after the first consumer has removed the last item, the second consumer tries to remove another item.

The first IVR found by Impara-IMC is depicted simplified on the right. The simplification merges all individual edges of a procedure into a single edge, which is possible as Impara-IMC does not apply context switches inside of procedures during the first iteration. Since both procedures appear to be executed atomically, no assertion violation is found during the first iteration. We ran the program with a program schedule corresponding to the first IVR. As expected, we have not observed any assertion violations.




### 5.4 Declarative synchronization

```
1  Variables:
2    int block
3    boolean busy
4    boolean inode
5    mutex m_inode
6    mutex m_busy
7  Initially: inode = busy
8  Thread T1:
9    while true:
10     lock(m_inode)
11     if not inode:
12       lock(m_busy)
13       busy := true
14       unlock(m_busy)
15       inode := true
16     block := 1
17     unlock(m_inode)
18 Thread T2:
19   while true:
20     lock(m_busy)
21     if not busy:
22       block := 0
23     unlock (m_busy)
24 Thread T3:
25   while true:
26     lock(m_inode)
27     lock(m_busy)
28     inode := false
29     busy := false
30     unlock(m_inode)
31     unlock(m_busy)
```

The above program shows an extension of a benchmark used in [15], which is a simplified extract of the multi-threaded Frangipani file system. The program uses a time-varying mutex: depending on the current value of the busy bit, a disk block is protected by m_busy or m_inode. We want to evaluate whether we can use Impara-IMC to generate safe program schedules even if all mutexes are (intentionally) removed from the program.

```
1  Thread T1:
2    while true:
3      if not inode:
4        busy := true
5        inode := true
6      atomic−begin
7      assume inode and busy
8      block := 1
9      atomic−end
```

```
10 Thread T2:
11   while true:
12     if not busy:
13       atomic−begin
14       assume not busy
15       block := 0
16       atomic−end
```

```
17 Thread T3:
18   while true:
19     atomic−begin
20     assume inode = busy
21     inode := false
22     busy := false
23     atomic−end
```

For this purpose, we use a variant of the file system benchmark where all mutexes are removed and synchronization constraints are declared as assume statements, shown above. It is sufficient to assure for $T_1$ that the block is written only if it is allocated, i.e., both inode and busy are true. For $T_2$, it is sufficient to assure that the block is only reset if it is not busy, i.e., busy = false. Finally, for $T_3$, it is necessary to assure that the block is deallocated only if it is already deallocated or fully allocated, i.e., inode = busy.

```
1 Thread T2':
2   while true:
3     atomic−begin
4     assume not busy
5     block := 0
6     atomic−end
```

Running Impara-IMC on the file system benchmark without mutexes yields a first program schedule that schedules $T_1$, $T_2$, $T_3$ repeatedly in this order, according to our simple heuristic for an initial IVR. However, although all executions permitted by this schedule are fair, the if-condition of $T_2$ always evaluates to false and $T_2$ never performs useful work. To obtain a more useful schedule, we inform the model checker that the (omitted) else-branch of Thread $T_2$ is not useful. We encode this information by inserting else: assume false. After simplifying the code, we obtain $T_2'$ as depicted on the right. For the updated code, Impara-IMC yields a first scheduler that schedules $T_3$ before $T_2$ before $T_1$, so that all threads perform useful work.

### 5.5 Performance

Tab. 1 shows the performance impact of enforcing IVRs on several correct programs. Each program is model-checked once until the first IVR (Impara-IMC) and once completely (Impara-C). As a baseline, the program is run without schedule enforcement (unconstrained). The first IVR is enforced without (Opt0), and with optimizations (Opt1, Opt2). Opt1 applies POR and omits operations on synchronization objects (mutexes, barriers).[4] Opt2 uses, in addition to Opt1, longer section schedules (by replicating a section eight times) and stronger partial-order reduction that identifies independent accesses to distinct indices of an array. Additionally, for the producer-consumer benchmark, we apply a compiler-like optimization, removing and reordering events to reduce the number of constraints.[5] Both Opt1 and Opt2 enable the concurrent execution of more memory accesses, e.g., because the beginning of a critical section can

[4] As enforcing an IVR is redundant to synchronization over existing mutexes and barriers, omitting them is safe.

[5] Opt2 follows a general algorithm, however we do not automate our implementation of Opt2, as it would be a large effort to implement compiler optimizations. Our implementation of Opt1 is automated.

Table 1: Experimental results (to: timeout, rounded to full seconds)
Performance is measured in number of useful (e.g., with a successful concurrent access such as a produced item) loop iterations within a time limit of 2 seconds.

| | Model checking | | Performance (higher is better) | | | |
|---|---|---|---|---|---|---|
| Benchmark | Time 1st IVR | Impara-C | Opt0 | Opt1 | Opt2 | Unconstr. |
| prod.-cons. 1p 1c | **2m 0s** | to (72h) | 4 864 489 | 7 466 093 | **11 370 258** | 8 199 202 |
| prod.-cons. 2p 2c | **23m 47s** | to (72h) | 3 400 187 | 5 959 041 | 8 428 598 | **11 643 208** |
| prod.-cons. 4p 4c | **4h 29m 53s** | to (72h) | 1 327 063 | 2 576 695 | 3 676 876 | **7 210 796** |
| double lock 1 ms | 0s | 0s | 1 845 | 1 834 | **3 217** | 1 797 |
| file system | 0s | 0s | 3 667 | 4 877 035 | 6 705 672 | **23 822 129** |
| barrier | **1s** | 4m 14s | 1 238 720 | 8 285 228 | **14 586 849** | 1 077 907 |

already be executed before a thread arrives at a constrained access that has to wait. The schedules for each benchmark (Opt0–Opt2) are obtained from the first IVR. As all benchmarks use unbounded loops, we measure the execution time performance by counting useful (i.e., with a successful concurrent access such as a produced item) loop iterations and terminating the execution after 2 seconds.

We use the producer-consumer implementation (with correct synchronization and buffer size 1000) from SV-COMP [1] (stack_safe), modified with an unbounded loop and with 1, 2, and 4 producers and consumers. The double lock benchmark is a corrected version (lock operations in $T_2$ reversed) of the deadlock benchmark (Sec. 5.2), where the critical section is simulated by sleeping for 1 ms; the uncorrected version reached a deadlock after only 172 loop iterations. The file system benchmark from SV-COMP (time_var_mutex_safe) is extended with a third thread and again with unbounded loops as in Sec. 5.4. The barrier benchmark uses two barriers to implement ring communication between threads.

As the model checking columns of Tab. 1 show, Impara-IMC finds the first IVR often much faster than or at least as fast as it takes Impara-C for complete model checking; it can produce an IVR even for our largest benchmarks, where Impara-C times out. For a buffer size of 5, Impara-C can verify the producer-consumer benchmark even with eight threads but again, Impara-IMC is considerably faster in finding the first IVR (cf. Appendix D). Subsequent IVRs were generated considerably faster than the first IVR, which might be caused by caching of facts in the model checker.

Somewhat surprisingly, some benchmarks are slower when executed unconstrained. We conjecture that this is caused by more memory accesses being executed in parallel under Opt2. In all but one cases, Opt2 is considerably faster than Opt1, which is considerably faster than Opt0. The highest overhead is observed for the file system benchmark, where Opt2 is about 3.5 times slower than the unconstrained execution. We conjecture that the high overhead here stems from an unequal distribution of loop iterations among threads, when executed unconstrained: the loop body of $T_2$ was executed nearly 100 times more frequently than $T_1$, while it is shorter and probably faster. Opt0–Opt2 execute all threads nearly balanced. In addition to the Pthread barriers used in the barrier benchmark, we tried a variant with busy waiting barriers, where the un-

constrained execution showed a performance of 13 567 135, which is still slower than Opt2. When the buffer size of the producer-consumer benchmark with eight threads is reduced to 5, the performance of unconstrained executions decreases to 3 240 136 compared to 3 392 111 with Opt2 (detailed results are in Appendix D).

Even in repeated executions of the experiment, the unconstrained variant of double lock showed only "starving" executions in the sense that the second thread was never able to acquire the mutexes before the timeout of 2 seconds. Hence, the constrained executions improve on the operating system scheduler in terms of a balanced execution of all threads.

In order to compare to the enforcement of *input-covering schedules* [7] (explained in Section 6), we measure the overhead of our scheduler implementation on the pfscan benchmark used there. Pfscan is a parallel implementation of grep and uses 1 producer and 2 consumer threads to distribute tasks, consisting of reading and searching a file for a given query. As input, we use 8 files with 100MB of random content each. We evaluate 4 different schedules[6], which show an overhead between 3% and 10% execution time overhead (with Opt2). Hence, IVRs can perform much better than input-covering schedules (60% overhead reported in [7]). Additional details are given in Appendix D.

## 6 Related work

Unbounded model checking [20,40,33,18] is a technique to verify the correctness of potentially non-terminating programs. In our setting, we deploy algorithms that use abstract reachability trees (ARTs) [22,29,40] to represent the already explored state space and schedules, and perform this exploration in a forward manner. Instead of discarding an ART after an unsuccessful attempt to verify a program, we use the ART to extract safe schedules.

Conditional model checking [8] reuses arbitrary intermediate verification results. In contrast to our approach, they are not guaranteed to prove the safety of a program that is functional under all inputs and does not enforce the preconditions (e.g., scheduling constraints) of the intermediate result.

Context bounding [37,36,32] eases the model checking problem by bounding the number of context switches. It is limited to finite executions and unlike our approach, does not enforce schedules at runtime.

Automated fence insertion [13,25,2,3,27] transforms a program that is safe under sequential consistency to a program that is also safe under weaker memory models. While the amount of non-determinism in the ordering of events is reduced, non-determinism due to scheduling can not be influenced. Synchronization synthesis [19] inserts synchronization primitives in order to prevent incorrect executions, but may introduce deadlocks.

Deterministic multi-threading (DMT) [4,6,7,12,11,28,31,35] reduces non-determinism due to scheduling in multi-threaded programs. Schedules are chosen dynamically, depending on the explicit input, and can not be enforced by a

[6] As Impara cannot handle several features used by pfscan (such as condition variables, structs, and standard output), we manually generate initial IVRs.

model checker. Nevertheless, there are combinations with model checking [11] and instances which schedule based on previously recorded executions [12].

We are aware of only one DMT approach that supports symbolic inputs [7]. Similar to our *sections*, *bounded epochs* describe infinite schedules as permutations of finite schedules. Via symbolic execution, an *input-covering* set of schedules is generated, which contains a schedule for each permutation of bounded epochs. As all permutations need to be analyzed (even if they are infeasible), state space explosion through concurrency is only partially avoided; indeed, the experimental evaluation shows that the analysis is infeasible even for five threads when the program has many such permutations. In contrast, we do not require race-freedom, use model checking, sections may contain multiple threads, omit infeasible schedules, and allow a safe execution from the first schedule on, i.e., an IVR can be considerably smaller than an input-covering set of schedules.

Deterministic concurrency requires a program to be deterministic regardless of scheduling. In [38], a deterministic variant of a concurrent program is synthesized based on constraints on conflicts learned by abstract interpretation. In contrast to DMT, symbolic inputs are supported, however no verification of general safety properties is done and the degree of non-determinism is not adjustable, in contrast to IVRs.

Sequentialized programs [37,26,14,23,33,34] emulate the semantics of a multi-threaded program, allowing tools for sequential programs to be used. The amount of possible schedules is either not reduced at all or similar to context bounding.

## 7 Conclusion

We present a formal framework for using IVRs to extract safe schedulers. We state why it is legitimate to constrain scheduling (in contrast to inputs) and formulate general requirements a model checker has to satisfy in our framework. We instantiate our framework with the Impact model checking algorithm and find in our evaluation that it can be used to 1. model check programs that are intractable for monolithic model checkers 2. safely execute a program, given an IVR, even if there exist unsafe executions 3. synthesize synchronization via assume statements 4. guarantee fair executions. A drawback of enforcing IVRs is a potential execution time overhead, however, in several cases, constrained executions turned out to be even faster than unconstrained executions.

## References


1. Benchmark suite of the competition on software verification (SV-COMP). `https://github.com/sosy-lab/sv-benchmarks`
2. Abdulla, P.A., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS. Springer (2012)
3. Abdulla, P.A., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: TACAS. LNCS, Springer (2013)
4. Aviram, A., Weng, S., Hu, S., Ford, B.: Efficient system-enforced deterministic parallelism. In: OSDI. USENIX Association (2010)
5. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
6. Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: Coredet: a compiler and runtime system for deterministic multithreaded execution. In: ASPLOS. ACM (2010)
7. Bergan, T., Ceze, L., Grossman, D.: Input-covering schedules for multithreaded programs. In: OOPSLA (2013)
8. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: FSE. ACM (2012)
9. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011)
10. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT **2**(3) (1999)
11. Cui, H., Simsa, J., Lin, Y., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G.A., Bryant, R.E.: Parrot: a practical runtime for deterministic, stable, and reliable threads. In: SOSP. ACM (2013)
12. Cui, H., Wu, J., Gallagher, J., Guo, H., Yang, J.: Efficient deterministic multithreading through schedule relaxation. In: SOSP. ACM (2011)
13. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: ICS. ACM (2003)
14. Fischer, B., Inverso, O., Parlato, G.: Cseq: A concurrency pre-processor for sequential C verification tools. In: ASE. IEEE (2013)
15. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: ESOP. LNCS, Springer (2002)
16. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL. ACM (2005)
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
18. Günther, H., Laarman, A., Sokolova, A., Weissenbacher, G.: Dynamic reductions for model checking concurrent software. In: VMCAI. LNCS, Springer (2017)
19. Gupta, A., Henzinger, T.A., Radhakrishna, A., Samanta, R., Tarrach, T.: Succinct representation of concurrent trace sets. In: POPL. ACM (2015)
20. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI. ACM (2004)
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70. ACM (2002)
22. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. ACM (2002)
23. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: CAV. Springer (2014)

24. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with wolverine. In: CAV. LNCS, vol. 6806, pp. 573–578. Springer (2011)
25. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: FMCAD. IEEE (2010)
26. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design (1) (2009)
27. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. LNCS, Springer (2013)
28. Liu, T., Curtsinger, C., Berger, E.D.: Dthreads: efficient deterministic multithreading. In: SOSP. ACM (2011)
29. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. LNCS, Springer (2006)
30. Metzler, P., Saissi, H., Bokor, P., Suri, N.: Quick verification of concurrent programs by iteratively relaxed scheduling. In: ASE. IEEE Computer Society (2017)
31. Mushtaq, H., Al-Ars, Z., Bertels, K.: Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In: High Performance Computing, Networking Storage and Analysis. IEEE (2012)
32. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI. ACM (2007)
33. Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for the safety verification of unbounded concurrent programs. In: ATVA. LNCS (2016)
34. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: ASE. IEEE Computer Society (2017)
35. Olszewski, M., Ansel, J., Amarasinghe, S.P.: Kendo: efficient deterministic multithreading in software. In: ASPLOS (2009)
36. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. LNCS, Springer (2005)
37. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI. ACM (2004)
38. Raychev, V., Vechev, M.T., Yahav, E.: Automatic synthesis of deterministic concurrency. In: SAS. Springer (2013)
39. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. Springer (1996)
40. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. In: FMCAD. IEEE (2013)

# A Formal model

## A.1 Basic definitions

**Definition 6 (program).** *A* program *is a tuple* $P = (S, s_{init}, Q, L, \mathfrak{l}_{error}, \mathscr{T})$*, where* $S$ *is the (potentially infinite) set of states,* $s_{init} \in S$ *is the unique initial state,* $Q$ *is the set of variables,* $L$ *is the set of local locations,* $\mathfrak{l}_{error}$ *is the unique error location, and* $\mathscr{T}$ *is a finite, totally-ordered set of threads* $T$*.*

**Definition 7 (location, state).** *The set of local locations is partitioned into a set of local locations* $L_T$ *for each thread* $T$*. A global location* $\mathfrak{l} \in L^{|\mathscr{T}|}$ *is a tuple of one local location for each thread, i.e.,* $\mathfrak{l} \in L_{T_1} \times \ldots L_{T_n}$*.*

*A state* $s \in S$ *is composed of a global location* $\mathfrak{l}(s)$ *and an interpretation of the variables* $Q$*, which maps variables to values.*

We assume that the location of the initial state is not the error location, i.e., $\mathfrak{l}(s_{init}) \neq \mathfrak{l}_{error}$. We write $s(v)$ for the value of variable $v$ in state $s$. We write $\mathfrak{l}_T(s)$ for the local location of thread $T$ at state $s$ and $\mathfrak{l}_T$ for the local location of thread $T$ in the global location $\mathfrak{l}$. Since local locations are disjoint, we also write $\mathfrak{l}(s)$ for $\mathfrak{l}_T(s)$ if $T$ is clear from the context. The variables $Q$ are partitioned into a set of global variables and a set of local variables for each thread. We write $Q_T$ for the union of the global variables and the local variables of thread $T$.

**Definition 8 (local and global transition relation).** *The global transition relation of a program* $P$ *is partitioned into a local transition relation* $R_T \subseteq S \times S$ *for each thread* $T \in \mathscr{T}$*. A thread may not change the local location and local variables of other threads, i.e., for all* $q \in Q \setminus Q_T$*, for all* $s, s' \in S$ *with* $R_T(s, s')$*, and for all threads* $T' \neq T$*, we require that* $s(q) = s'(q)$ *and* $\mathfrak{l}_{T'}(s) = \mathfrak{l}_{T'}(s')$*.*

**Definition 9 (transition, guard).** *A thread's local transition relation* $R_T$ *is partitioned into (local)* transitions $R_{\mathfrak{l},\mathfrak{l}'} \subseteq R_T$ *such that for all states* $s, s'$*:*

$$R_{\mathfrak{l},\mathfrak{l}'}(s, s') \Leftrightarrow (R_T(s, s') \wedge \mathfrak{l} = \mathfrak{l}_T(s) \wedge \mathfrak{l}' = \mathfrak{l}_T(s'))$$

*The* guard *of a transition* $R$*, written* $Guard(R)$*, encodes the predicate* $\exists s'.\, R(s, s')$*.*

We write $\mathscr{F}(Q)$ for the set of all first order formulae over the variables $Q$ and optional additional interpreted symbols. A *state formula* is a formula $\phi \in \mathscr{F}(Q)$. We assume that each transition $R$ can be represented by a *transition formula* $R_1 \wedge R_2$ such that $R_1 \in \mathscr{F}(Q)$ represents the guard $Guard(R)$ and $R_2 \in \mathscr{F}(Q \cup Q')$ represents the relation between state and successor state.

**Definition 10 (active and enabled transition).** *Let* $R_{\mathfrak{l},\mathfrak{l}'}$ *be a transition of a thread* $T$*.* $R_{\mathfrak{l},\mathfrak{l}'}$ *is* active *at global location* $\mathfrak{l}$ *and local location* $\mathfrak{l}_T$*.* $R_{\mathfrak{l},\mathfrak{l}'}$ *is* enabled *in those states* $s$ *that satisfy* $\mathfrak{l}_T(s) = \mathfrak{l}$ *and* $Guard(R_{\mathfrak{l},\mathfrak{l}'})$*. We require that there exists at most one enabled transition for a given thread and state.*

We write $\mathit{Transitions}(\mathfrak{l}_T) := \{R_{\mathfrak{l},\mathfrak{l}'} \subseteq R_T : \mathfrak{l} = \mathfrak{l}_T\}$ for the set of active transitions of $T$ at $\mathfrak{l}_T$. We write $\mathit{Next\text{-}Transition}(s, T)$ for the (unique) enabled transition of $T$ in $s$ if it exists and otherwise $\mathit{Next\text{-}Transition}(s, T) = \bot$. For a state $s$, we write $\mathit{enabled}(s) = \bigcup_{T \in \mathscr{T}} \mathit{Next\text{-}Transition}(s, T)$ for the set of enabled transitions of all threads in $s$.

Although $\mathit{Next\text{-}Transition}(s, T) = R$ is unique (if it exists), it can be non-deterministic, i.e., there may exist multiple successor states $s'$ with $R(s, s')$ and the program input decides which successor state to take.

Executions are sequences of states, interleaved with threads. In order to simplify the presentation, we assume that only infinite executions are desired.

**Definition 11 (execution).** *An* execution $\tau$ *of a program is a sequence* $s_0, T_1, s_1, \ldots$ *such that* $s_0$ *is the initial state of the program and for every adjacent triple* $(s_i, T_i, s_{i+1})$ *in the sequence,* $s_i$ *and* $s_{i+1}$ *are related by the global transition relation. If* $\tau$ *is finite, it is of the form* $\tau = s_0, T_1, s_1, \ldots, T_n, s_n$ *and additionally* $\mathit{enabled}(s_n) = \emptyset$ *holds. An execution is* safe *if it does not reach the error location, i.e.,* $\mathfrak{l}(s_i) \neq \mathfrak{l}_{error}$ *for* $0 \leq i \leq n$.

Safety properties can be encoded directly into a program (via the error location) so that executions satisfy the safety property if and only if they are safe. To simplify the presentation and without loss of generality, we assume that only a single error location exists. Multiple error locations can be modeled by a single error location and additional transitions to this location.

**Definition 12 (deadlock).** *A* deadlock *is a state* $s$ *with active transitions but without enabled transitions, i.e.,* $\bigcup_{T \in \mathscr{T}} \mathit{Transitions}(\mathfrak{l}_T(s)) \neq \emptyset \wedge \mathit{enabled}(s) = \emptyset$.

As deadlocks are prominent issues in concurrent programming, we assume that deadlocks are always undesirable, whether or not they are explicitly marked as errors by the safety property. In order to simplify the presentation, we assume that there are no finite, desired, executions, i.e., all finite executions lead to a deadlock. In other words, we assume that there exists an active (but not necessarily enabled) transition in all states (this transition can be a dummy transition if a program intentionally terminates).

**Definition 13 (blocking transitions).** *Deadlocks may only arise through blocking transitions (such as a lock acquisition of an already taken lock). Formally,* $T$ may block *at a location* $\mathfrak{l}_T$ *if there exist states* $s, s'$ *with this location* $\mathfrak{l}_T = \mathfrak{l}_T(s) = \mathfrak{l}_T(s')$ *such that* $T$ *has an enabled transition at* $s$ *but no enabled transition at* $s'$*, i.e.,* $\mathit{Next\text{-}Transition}(s, T) \neq \bot \wedge \mathit{Next\text{-}Transition}(s', T) = \bot$.

We assume that such pairs of a thread and location are marked with a predicate $\mathit{may\text{-}block}(\mathfrak{l}_T)$. It is permitted to overapproximate the predicate, at the expense of performance, i.e., we require that for all pairs $(\mathfrak{l}_T, T)$ which may block that $\mathit{may\text{-}block}(\mathfrak{l}_T)$ holds but not the converse. Furthermore, we assume that threads do not block at control flow branchings, i.e., for all threads $T$ and locations $l$ with $\mathit{may\text{-}block}(\mathfrak{l}_T)$, $|\mathit{Transitions}(\mathfrak{l}_T)| = 1$. This requirement can easily be satisfied by splitting transitions that model both a lock acquisition and a control flow branching into two separate transitions.

Beyond deadlocks, we assume that unfair executions are undesirable.

**Definition 14 (fair execution).** *An execution $\tau$ is* fair *if every process that is enabled infinitely often along $\tau$ is scheduled infinitely often along $\tau$. We write Fair-Executions$(P)$ for the fair executions of a program $P$.*

Intuitively, non-determinism can arise through scheduling of threads and through (non-deterministic) inputs, which can be modeled as multiple transitions of a thread that are enabled at a state. A *scheduler* can resolve the former kind of non-determinism.

**Definition 15 (scheduler).** *A* scheduler $\zeta : (S \times \mathscr{T})^* \times S \to \mathscr{T}$ *of a program $P$ is a function from finite sequences of states and threads to threads such that for all sequences $\tau = s_0, T_1, \ldots, T_n, s_n$, $\zeta$ chooses a thread with an enabled transition at $s_n$ if such a thread exists, i.e., enabled$(s_n) \neq \emptyset \Rightarrow$ Next-Transition$(s_n, \zeta(\tau)) \neq \bot$.*

We write *Schedulers*$(P)$ for the set of all schedulers of $P$. We write *Executions*$(P, \zeta)$ for the set of all executions $\tau = s_0, T_1, s_1, \ldots$ of $P$ such that for each adjacent triple $(s_i, T_{i+1}, s_{i+1})$, $T_{i+1} = \zeta(s_0, T_1, \ldots, s_i)$. If $\tau = s_0, T_1, \ldots, T_n, s_n$ is finite, additionally *enabled*$(s_n) = \emptyset$ must hold.

**Definition 16 (deadlock-free and fair scheduler).** *A scheduler $\zeta$ for a program $P$ is* deadlock-free*, written deadlock-free$(\zeta)$, if no execution in Executions$(P, \zeta)$ reaches a deadlock and $\zeta$ is* fair *if all executions in Executions$(P, \zeta)$ are fair, i.e., Executions$(P, \zeta) \subseteq$ Fair-Executions$(P)$.*

It is important to note that unless the program is in a terminal state (no transitions are enabled), a scheduler must schedule a thread that has an enabled transition in that state. A scheduler is deadlock-free if it can always find a deadlock-free execution, even if the program shows executions that reach a deadlock.

Program inputs are modeled as follows. A program has an input alphabet $X$ and each input symbol $\iota \in X$ makes transitions deterministic, i.e., for each transition $R$, there exists a function $R^\iota : S \to S$ such that for all states $s$ at which $R$ is enabled, $R^\iota(s) = s'$ for some $s'$ with $R(s, s')$. Non-deterministic input, or, more generally, influence from the environment, can be modeled by an *input* (as a dual concept to schedulers), defined as follows.

**Definition 17 (input).** *An* input *is a function $\chi : (S \times \mathscr{T})^* \to X$, which chooses an input symbol depending on the current execution prefix.*

In conjunction, an input and a scheduler render a program completely deterministic: the execution of $P$ under input $\chi$ and scheduler $\zeta$ is the unique execution $s_0, T_1, s_1, \ldots \in$ *Executions*$(P, \zeta)$ such that for all adjacent triples $(s_i, T_{i+1}, s_{i+1})$, for *Next-Transition*$(s_i, T) = a$, and for $\chi(s_0, T_1, \ldots, s_i, T_{i+1}) = \iota$, $s_{i+1} = a^\iota(s_i)$.

For partial order reduction, we state the following requirements on independence relations on transitions, which induce an equivalence relation on executions (Mazurkiewicz traces).

**Definition 18 (independence).** *An independence relation $\parallel$ is a symmetric relation on transitions such that for any two threads $T_1, T_2$ and any two transitions $R_1 \in R_{T_1}$ and $R_2 \in R_{T_2}$, $R_1 \parallel R_2$ implies*

- $T_1 \neq T_2$
- $\forall \iota_1, \iota_2 \in X. \forall s \in S. R_1, R_2 \in \mathit{enabled}(s) \Rightarrow R_1^{\iota_1}(R_2^{\iota_2}(s)) = R_2^{\iota_1}(R_1^{\iota_2}(s))$
- $\forall \iota \in X. \forall s \in S. s_1 \in \mathit{enabled}(s) \Rightarrow (R_2 \in \mathit{enabled}(s) \Leftrightarrow R_2 \in \mathit{enabled}(R_1^{\iota}(s))))$

We write $R_1 \nparallel R_2$ if $R_1$ and $R_2$ are not independent, i.e., dependent.

### A.2 Requirements on incomplete verification results

**Definition 19 (incomplete verification result).** *An* incomplete verification result *(IVR) for a program $P$ is a function $\mathcal{R} : (S \times \mathscr{T})^* \times S \to \mathcal{P}(\mathscr{T})$ that maps execution prefixes to sets of threads.*

An IVR represents scheduling constraints. We write $\mathit{Schedulers}(P_{\mathcal{R}})$ for the set of schedulers that enforce these scheduling constraints: for all $\zeta_{\mathcal{R}} \in \mathit{Schedulers}(P_{\mathcal{R}})$ and for all execution prefixes $\tau = s_o, T_1, s_1, \ldots, s_n$, we have $\zeta_{\mathcal{R}}(\tau) \in \mathcal{R}(\tau)$, i.e., $\mathcal{R}(\tau)$ specifies a set of threads that are permitted to be scheduled after $\tau$, according to the scheduling constraints. We write $\mathit{Executions}(\mathcal{R})$ for the set of executions that are *permitted* by $\mathcal{R}$, defined as $\mathit{Executions}(\mathcal{R}) := \bigcup_{\zeta_{\mathcal{R}} \in \mathit{Schedulers}(P_{\mathcal{R}})} \mathit{Executions}(P, \zeta_{\mathcal{R}})$.

#### Safety

**Definition 20 (safe incomplete verification result).** *An incomplete verification result $\mathcal{R}$ is* safe *if all executions in $\mathit{Executions}(\mathcal{R})$ are safe.*

An unsafe incomplete verification result permits an unsafe execution and is called a *counterexample*.

#### Completeness

**Definition 21 (realizable incomplete verification result).** *A safe incomplete verification result is* realizable *if there exists a scheduler such that all executions possible under this scheduler are permitted by $\mathcal{R}$, i.e., $\mathit{Schedulers}(P_{\mathcal{R}}) \neq \emptyset$.*

**Definition 22 (deadlock-free incomplete verification result).** *A realizable incomplete verification result $\mathcal{R}$ is* deadlock-free *if all schedulers which enforce $\mathcal{R}$ are deadlock-free, i.e., $\mathit{Schedulers}(P_{\mathcal{R}}) \neq \emptyset \wedge \forall \tau \in \mathit{Executions}(P_{\mathcal{R}}). \mathit{deadlock\text{-}free}(\tau)$.*

Equivalently, the requirements on $\mathcal{R}$ can be defined by the following game: there are two players, the scheduler player and the input player. Configurations are prefixes $\tau$ of executions. If $\tau$ is of the form $s_0, T_1, s_1, \ldots, T_n, s_n$, the scheduler player appends a thread $T$ such that $\mathit{Next\text{-}Transition}(s_n, T) \neq \bot$. If $\tau$ is of the form $s_0, T_1, s_1, \ldots, T_n$, the input player appends a state $s_n$ such that $R(s_{n-1}, s_n)$, where $R = \mathit{Next\text{-}Transition}(s_{n-1}, T)$. The scheduler player wins if an error-free terminal state that is not a deadlock is reached or if the resulting execution is infinite. Consequently, the input player wins if an error state is reached. $\mathcal{R}$ is an appropriate incomplete verification result if it corresponds to a winning strategy of the scheduler player.

**Fairness**

**Definition 23 (incomplete verification result that admits fairness).** *A deadlock-free* $\mathcal{R}$ admits fairness *if there exists a fair scheduler* $\zeta \in \mathit{Schedulers}(P_{\mathcal{R}})$.

**Definition 24 (fair incomplete verification result).** *A deadlock-free incomplete verification result* $\mathcal{R}$ *is* fair *if all schedulers* $\zeta \in \mathit{Schedulers}(P_{\mathcal{R}})$ *are fair.*

### A.3 Abstract reachability trees as incomplete verification results

**Definition 25 (abstract reachability tree [29,40]).** *An* abstract reachability tree *(ART) is a tuple* $\mathscr{A} = (V_{\mathscr{A}}, \epsilon, \rightarrow_{\mathscr{A}}, \rhd)$ *such that* $(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ *is a finite tree with root* $\epsilon \in V_{\mathscr{A}}$ *and* $\rhd \subseteq V_{\mathscr{A}} \times V_{\mathscr{A}}$ *is a covering relation. Nodes* $v$ *are labeled with global locations and state formulas, written* $\mathfrak{l}(v)$ *and* $\phi(v)$*, respectively. Edges* $(v, w) \in \rightarrow_{\mathscr{A}}$ *are labeled with a thread and a transition formula, written* $(v, T, R, w) \in \rightarrow_{\mathscr{A}}$*. We also write* $v \xrightarrow{T,R}_{\mathscr{A}} w$ *or* $v \xrightarrow{T}_{\mathscr{A}} w$ *to express that such an edge exists. If there exists* $(v, w) \in \rhd$*, we write* $\mathit{covered}(v)$.

For ease of notation, we do not distinguish between a transition and its transition formula. An ART is *well-labeled* if:

- $\phi(\epsilon)$ represents the initial state,
- for all states $s, s'$ and for all edges $v \xrightarrow{T,R_{\mathfrak{l},\mathfrak{l}'}}$ in $\mathscr{A}$: $(\phi(v)(s) \wedge R_{\mathfrak{l},\mathfrak{l}'}(s, s')) \Rightarrow \phi(w)(s')$, and
- for every $v, w$ with $v \rhd w$: $\phi(v) \Rightarrow \phi(w)$ and $\neg \mathit{covered}(w)$.

**Schedulers induced by abstract reachability trees** A well-labeled ART $\mathscr{A} = (V_{\mathscr{A}}, \epsilon, \rightarrow_{\mathscr{A}}, \rhd)$ directly corresponds to an IVR $\mathcal{R}_{\mathscr{A}} : (S \times \mathscr{T})^* \times S \rightarrow \mathcal{P}(\mathscr{T})$ that simulates an execution by traversing $\mathscr{A}$. Before we define $\mathcal{R}_{\mathscr{A}}$, we introduce a correspondence relation between executions and paths in $\mathscr{A}$.

When a path reaches a covered node, we continue the path at the covering node, so that we can match infinite executions. For a direct correspondence of an execution to such a path, we skip covered nodes. Formally, for a path $\pi$ along $\rightarrow_{\mathscr{A}}$ and $\rhd$ edges, we write $\hat{\pi}$, for the unique path along $\rightarrow_{\mathscr{A}}$ edges that results from replacing all $v_1 \xrightarrow{T,R}_{\mathscr{A}} v_2 \rhd v_3$ edges in $\pi$ by $v_1 \xrightarrow{T,R}_{\mathscr{A}} v_3$. NB, $\hat{\pi}$ is not necessarily a path in $\mathscr{A}$.

An execution $\tau = s_0, T_1, s_1, \ldots$ *corresponds* to a path $\pi$ in $\mathscr{A}$, written $\tau \sim \pi$, if for $\hat{\pi} = v_0, T_1', R_1, v_1, \ldots$ and for all $i > 0$: $v_0 = \epsilon$ and $T_i = T_i'$ and $s_i \vDash \phi(v_i)$ and $R(s_{i-1}, s_i)$.

Based on this correspondence relation, we define $\mathcal{R}_{\mathscr{A}}$.

**Definition 26 (incomplete verification result induced by an abstract reachability tree).** *The IVR* $\mathcal{R}_{\mathscr{A}}$ *represented by a well-labeled abstract reachability tree* $\mathscr{A}$ *is defined as follows. Let* $\tau = s_0, T_1, s_1, \ldots, s_n$ *be an execution prefix. If* $\mathscr{A}$ *contains no path that corresponds to* $\tau$*,* $\mathcal{R}_{\mathscr{A}}(\tau) := \mathscr{T}$*. Otherwise, let* $\pi = v_0, T_1', R_1, v_1, \ldots, v_n$ *be the path in* $\mathscr{A}$ *that corresponds to* $\tau$*.* $\mathcal{R}_{\mathscr{A}}(\tau) := \mathscr{T}' \subseteq \mathscr{T}$ *such that* $\mathscr{T}'$ *is the set of threads that are expanded at* $v_n$ *(in case* $v_n$ *is covered by some node* $w$*,* $\mathscr{T}'$ *is the set of threads that are expanded at* $w$*).*

### Safety

**Definition 27 (safe abstract reachability tree).** *An ART is* safe *if all nodes $v$ that are labeled with the error location are also labeled with $\phi(v) = \mathit{false}$.*

### Completeness

**Definition 28 (deadlock-free ART).** *A well-labeled, safe ART $\mathscr{A}$ is* deadlock-free *if each node $v$ of $\mathscr{A}$ is either covered or one thread $T$ is fully expanded at $v$ and for all edges $w \xrightarrow{T,R}_{\mathscr{A}} w'$ in $\mathscr{A}$ where $R$ may block, $\phi(w)$ can prove the feasibility of $a$, i.e.:*

$$\begin{aligned}\forall v \in V_{\mathscr{A}}.\, (\mathit{covered}(v) \vee \exists T \in \mathscr{T}.\, \forall (l, R, l') \in \mathit{Transitions}(\mathfrak{l}(v))T.\, \exists w \in V_{\mathscr{A}}.\, v \xrightarrow{T,R}_{\mathscr{A}} w)\\ \wedge\, \forall v, w \in V_{\mathscr{A}}.\, \forall T \in \mathscr{T}.\, \forall a = (l, R, l') \in R_T.\, ((v \xrightarrow{T,R}_{\mathscr{A}} w \wedge \mathit{may\text{-}block}(\mathfrak{l}_T))\\ \Rightarrow (\phi(v) \Rightarrow \mathit{Guard}(R)))\end{aligned}$$

The requirement $\phi(v) \Rightarrow \mathit{Guard}(R)$, i.e., to require that $\phi(v)$ can prove the feasibility of $R$, may seem strong and an ART that satisfies this constraint difficult to construct. To limit the associated cost, we require this feasibility check only for transitions that may block. Furthermore, in Appendix E, we argue that such an ART is easy to construct for "reasonable" programs.

**Lemma 6.** *For all deadlock-free ARTs $\mathscr{A}$, $\mathcal{R}_{\mathscr{A}}$ is a deadlock-free verification result.*

*Proof. Let $\mathcal{R}_{\mathscr{A}}$ be the IVR of a deadlock-free ART $\mathscr{A}$. First, we construct a scheduler that enforces $\mathcal{R}_{\mathscr{A}}$, which proves that $\mathcal{R}_{\mathscr{A}}$ is realizable. Second, we show that all schedulers that enforce $\mathcal{R}_{\mathscr{A}}$ are deadlock-free, which concludes the proof that $\mathcal{R}_{\mathscr{A}}$ is deadlock-free.*

*Let $\mathscr{T}'(\tau) = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T \in \mathscr{T} : \mathit{Next\text{-}Transition}(s_n, T) \neq \bot\}$. Let $\zeta : (S \times \mathscr{T})^* \times S \to \mathscr{T}$ be an arbitrary function such that $\forall \tau.\, \zeta(\tau) \subseteq \mathscr{T}'(\tau)$ whenever $\mathscr{T}'(\tau)$ is not empty. (A description of how $\zeta$ can be constructed is given by the definition of $\mathcal{R}_{\mathscr{A}}$.) By construction, $\zeta$ enforces $\mathcal{R}_{\mathscr{A}}$ if $\zeta$ is a scheduler. We show that $\zeta$ is a scheduler by contradiction. Assume that $\zeta$ is not a scheduler. Then there exists an execution prefix $\tau = s_0, T_1, s_1, \ldots, s_n$ such that $\zeta(\tau) = T$, $\mathit{Next\text{-}Transition}(s_n, T) = \bot$ and $\mathit{enabled}(s_n) \neq \emptyset$.*

**case $\tau$ does not correspond to a path in $\mathscr{A}$:** *By the definition of $\mathcal{R}_{\mathscr{A}}$, $\mathcal{R}_{\mathscr{A}}(\tau) = \mathscr{T}$. By assumption $\mathit{enabled}(s_n) \neq \emptyset$, $\mathscr{T}'$ is not empty. By the construction of $\zeta$, $T \in \mathscr{T}'$. Contradiction to $\mathit{Next\text{-}Transition}(s_n, T) = \bot$.*

**case $\tau$ corresponds to a path $\pi = v_0, T_1, R_1, v_1, \ldots, v_n$ in $\mathscr{A}$:** *By the construction of $\mathcal{R}_{\mathscr{A}}$, $T$ is expanded at $v_n$.*

**case** *may-block*$(\mathfrak{l}_T(v_n))$**:** *By the definition of* may block*, $T$ has exactly one transition $R$ active at $\mathfrak{l}_T(v_n)$. As $\mathscr{A}$ is deadlock-free, $\phi(v_n) \Rightarrow \mathit{Guard}(R)$. By assumption $\tau \sim \pi$, $s_n \vDash \phi(v_n)$. Hence, $\phi(v_n) \vDash \mathit{Guard}(R)$ and $R \in \mathit{enabled}(s_n)$. Contradiction to $\mathit{enabled}(s_n) = \emptyset$.*

**case not** *may-block*$(\mathfrak{l}_T(v_n))$**:** *By the definition of* may block*,* *Next-Transition*$(s_n, T) = R \neq \bot$ *for some transition* $R$*. Contradiction to* *Next-Transition*$(s_n, T) = \bot$*.*

*It remains to show that all schedulers that enforce* $\mathcal{R}_{\mathscr{A}}$ *are deadlock-free. Let* $\zeta$ *be an arbitrary scheduler that enforces* $\mathcal{R}_{\mathscr{A}}$*. Assume that* $\zeta$ *is not deadlock-free. Then there exists an execution* $\tau = s_0, T_1, s_1, \ldots, s_n \in$ *Executions*$(P, \zeta)$ *such that* $s_n$ *is a deadlock, i.e.,* *enabled*$(s_n) = \emptyset \wedge \exists T \in \mathscr{T}.$ *Next-Transition*$(s_n, T) \neq \bot$*. As* $\tau \in$ *Executions*$(\mathcal{R}_{\mathscr{A}})$*,* $\tau$ *corresponds to a path* $\pi = v_0, T_1, R_1, v_1, \ldots, v_n$ *in* $\mathscr{A}$*. Let* $T = \zeta(\tau)$*. By choice of* $\zeta$*,* $T$ *is expanded at* $v_n$*. With the same argument as above, in case* *may-block*$(\mathfrak{l}_T(v_n))$*, we have* $\phi(v_n) \Rightarrow$ *Guard*$(R)$ *for some transition* $R \in$ *Transitions*$(\mathfrak{l}_T(v_n))$ *and a contradiction to* *enabled*$(s_n) = \emptyset$ *and in case not* *may-block*$(\mathfrak{l}_T(v_n))$*, we have* *Next-Transition*$(s_n, T) \neq \bot$ *and a contradiction to* *Next-Transition*$(s_n, T) = \bot$*.*

### Fairness

**Definition 29 (cycle).** *Given an ART* $\mathscr{A}$*, a* $(\rhd \cup \rightarrow_{\mathscr{A}})$*-cycle is a simple cycle in the graph* $(V_{\mathscr{A}}, \rhd \cup \rightarrow_{\mathscr{A}})$*, i.e., a finite sequence of nodes* $v_1, \ldots, v_n$ *such that:*

- $v_1 = v_n$
- $v_i \neq v_j$ *for all* $i, j \in \{2, \ldots, n-1\}, i \neq j$
- $v_i \rhd v_{i+1}$ *or* $v_i \rightarrow_{\mathscr{A}} v_{i+1}$ *for all* $i \in \{1, \ldots, n-1\}$

**Definition 30 (ART that admits fairness).** *Let* $\mathscr{A} = (V_{\mathscr{A}}, \rhd, \epsilon, \rightarrow_{\mathscr{A}})$ *be a deadlock-free ART.* $\mathscr{A}$ admits fairness *if for every* $(\rhd \cup \rightarrow_{\mathscr{A}})$*-cycle* $c$*:*

$$\forall T \in \mathscr{T}.\, (\exists v' \in c.\, \textit{Transitions}(\mathfrak{l}_T(v')) \neq \emptyset \Rightarrow \exists v \in c.\, \exists w \in V_{\mathscr{A}}.\, \forall R \in \textit{Transitions}(\mathfrak{l}_T(v)).\, v \xrightarrow{T,R}_{\mathscr{A}} w)$$

**Lemma 7.** *For all ARTs* $\mathscr{A}$ *that admit fairness,* $\mathcal{R}_{\mathscr{A}}$ *is an incomplete verification result that admits fairness.*

*Proof. We need to show that there exists a fair scheduler* $\zeta$ *that enforces an arbitrary ART* $\mathscr{A}$ *that admits fairness. After constructing* $\zeta$*, we show that* $\zeta$ *is fair by contradiction.*

*Let* $\tau = s_0, T_1, s_1, \ldots, s_n$ *be an execution prefix and let* $\pi$ *be a path such that* $\tau$ *corresponds to* $\pi = v_0, T_1, \ldots, v_n$*. By* $\gamma(T)$*, we denote the number of occurrences of* $T$ *in* $\pi$*. Let* $\mathscr{T}'$ *be the set of threads that is both enabled at* $s_n$ *and permitted by* $\mathscr{A}$*, i.e.,* $\mathscr{T}' = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T :$ *Next-Transition*$(s_n, T) \neq \bot\}$*. We let* $\zeta$ *schedule an arbitrary thread* $T \in \mathscr{T}'$ *such that no other thread in* $\mathscr{T}'$ *occurs less often in* $\pi$*, i.e.,* $\zeta(\tau) = T \in \mathscr{T}'$ *such that* $\forall T' \in \mathscr{T}'.\, \gamma(T) \leq \gamma(T')$*. By Lemma 6 and as* $\mathscr{A}$ *admits fairness,* $\zeta$ *is indeed a scheduler (*$\mathscr{T}'$ *is only empty when* *enabled*$(s_n)$ *is empty).*

*It remains to show that* $\zeta$ *is fair, i.e., that every execution scheduled by* $\zeta$ *is fair. Let* $\tau$ *be an execution that is scheduled by* $\zeta$ *(*$\tau$ *is of the form* $\tau = s_{init}, \zeta(s_{init}), s_1, \ldots$*). If* $\tau$ *is finite, it is trivially fair. Otherwise, assume that*

*$\tau$ is not fair. Then there exists a thread $T$ that is infinitely often enabled in $\tau$ but does not occur in $\tau$ after some prefix of $\tau$. Let $\pi$ be a path in $\mathscr{A}$ such that $\tau$ corresponds to pi. Let $v_T$ be a node at which $T$ is enabled and that occurs infinitely often in $\pi$. As $\mathscr{A}$ is finite and by Lemma 12 (p. 32), there exists a cycle that contains $v_T$ such that $\pi$ visits all nodes in this cycle infinitely often. As $\mathscr{A}$ admits fairness, there exists $v \xrightarrow{T,a}_{\mathscr{A}} v'$ such that $v$ is in this cycle and $a \in enabled(s)$ for all states $s$ that correspond to $v$. As $T$ is not scheduled in $\tau$ after some finite number $i$ of steps, there exist one or more other threads $T' \neq T$ with $v \xrightarrow{T'}_{\mathscr{A}} w$ for some $w \neq v'$ which are scheduled at $v$ for all steps $k > i$. Let $t$ be the set of those threads $T'$. By the construction of the scheduler, $\gamma(T') \leq \gamma(T)$ for all $T' \in t$. After only finitely many steps $l$, $\gamma(T) < \gamma(T')$ for all $T' \in t$ (e.g., take $l$ to be the product of the maximum path length from $v$ to $v$ and the number $\sum_{T' \in t} 1 + \gamma(T) - \gamma(T')$ of required visits of $v$). Hence, there exists a prefix of $\pi$ of length $l' \geq l$ in which $v \xrightarrow{T}_{\mathscr{A}} v'$ is the last step, i.e., $T$ has been scheduled. Contradiction to the assumption that $T$ is not scheduled after $i$ steps in $\pi$.*

**Definition 31 (fair ART).** *Let $\mathscr{A} = (V_{\mathscr{A}}, \rhd, \epsilon, \rightarrow_{\mathscr{A}})$ be a deadlock-free ART. $\mathscr{A}$ is* fair *if for every $(\rhd \cup \rightarrow_{\mathscr{A}})$-cycle $c$, $fair(c)$, where:*

$$fair(c) \equiv \forall T \in \mathscr{T}.\, (\exists v' \in c.\, Transitions(\mathfrak{l}(v'))T \neq \emptyset \Rightarrow \exists v \in c.\, \underline{\exists w \in c}.\, \forall a = (l, R, l') \in Transitions(\mathfrak{l}(v))T.\, v \xrightarrow{T,R}_{\mathscr{A}} w)$$

Note the difference between an ART that admits fairness and a fair ART (highlighted in the formula above): the successor node $w$ of $v$ that guarantees that a thread can be scheduled is required to be within the given cycle for fair ARTs.

**Lemma 8 (fairness).** *For all fair ARTs $\mathscr{A}$, $\mathcal{R}_{\mathscr{A}}$ is a fair verification result.*

*Proof. Let $\mathscr{A}$ be a fair ART. By Lemma 6 and as $\mathscr{A}$ is deadlock-free, there exists a scheduler $\zeta$ that enforces $\mathscr{A}$. It remains to show that $\zeta$ is fair, which we prove by contradiction. Suppose that an unfair execution $\tau$ is possible under $\zeta$. There exists a thread $T$ that is enabled infinitely often in $\tau$ but does not occur in $\tau$ after a finite prefix. Let $\pi$ be a path through $\mathscr{A}$ such that $\tau$ corresponds to $\pi$. As $V_{\mathscr{A}}$ is finite, there exists a node $v$ that occurs infinitely often in $\pi$ and at which $T$ is enabled. By Lemma 12 (p. 32), $v$ is part of a cycle of which all nodes occur infinitely often in $\pi$. By fairness, one edge in this cycle is labeled with $T$. By the definition of ARTs ($(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ is a tree), this edge occurs infinitely often in $\pi$. Contradiction.*

Given an ART $\mathscr{A}$ that admits fairness, one can generate a fair ART $\mathscr{A}'$ such that $Executions(\mathcal{R}_{\mathscr{A}'}) \subseteq Executions(\mathcal{R}_{\mathscr{A}})$. An algorithm that generates $\mathscr{A}'$ is given as Algorithm 4 in Appendix G.

## B Partial-order reduction

Intuitively, two executions $\tau, \tau'$ are equivalent if $\tau'$ can be obtained from $\tau$ by repeatedly swapping two adjacent, independent transitions (which corresponds

to Mazurkiewicz equivalence [17]). In order to match our definition of executions (which do not contain transitions explicitly), one has to augment each occurrence $s \xrightarrow{T} s'$ of a thread with the unique enabled transition $\mathit{Next\text{-}Transition}(s, T)$. The states occurring in $\tau'$ may differ from those in $\tau$, however, it is guaranteed that the values of local and global variables of each thread, $V_T$, are equal in the states in $\tau$ and $\tau'$ [17]. We make this notion of equivalence precise using *event sequences*, defined below.

**Definition 32 (branching node).** *A node $v$ in an ART $\mathscr{A}$ represents a branching, written $\mathit{branching}(v)$, if a single thread has at least two outgoing edges at $v$, i.e., $\exists T \in \mathscr{T}.\, \exists w, w' \in V_{\mathscr{A}}.\, w \neq w' \wedge v \xrightarrow{T} w \wedge v \xrightarrow{T} w'$.*

For an optimized partial-order reduction, successor nodes v with $\phi(v) \not\equiv \mathit{false}$ can be discarded.

**Definition 33 (section path).** *Given a node $v$ of an ART $\mathscr{A}$, a* section path from $v$ *is a finite sequence $v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n$ such that*

- $v_0 = v$
- $\forall i \in \{0, \ldots, n-1\}.\, (v_i \xrightarrow{T_i, R_i}_{\mathscr{A}} v_{i+1} \vee (\exists v'.\, v_i \xrightarrow{T_i}_{\mathscr{A}} v' \wedge (v', v_{i+1}) \in \triangleright))$
- $\forall i \in \{1, \ldots, n-1\}.\, \neg \mathit{branching}(v_i)$

We write $v \xrightarrow{\pi}_{\mathscr{A}} w$ if there exists a section path $\pi$ from $v$ to $w$ in $\mathscr{A}$.

**Definition 34 (event sequence).** *The* event sequence *of a section path $v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n$ is defined as $(T_0, k_0) \ldots (T_{n-1}, k_{n-1})$, where $k_i$ is the number of occurrences of $T_i$ in $T_0 \ldots T_{i-1}$.*

For $e_i = (T_i, k_i)$, we define $\mathit{tid}(e_i) = T_i$.

For an optimized partial-order reduction, adjacent transitions of a thread can be merged into a single event as long as the resulting event contains at most one global memory access.

The *Section schedule* of a section path describes the equivalence class of all executions that follow the section path. Hence, for a fixed state at the beginning of the section, each thread reads and writes the same values in all executions of the section that adhere to the section schedule.

**Definition 35 (section schedule).** *The* section schedule *of a section path $v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n$ with event sequence $e_0 \ldots e_{n-1}$ is the smallest partial order $\sigma = (V_\sigma, \rightarrow_\sigma)$ such that $V_\sigma = \{e_0, \ldots, e_{n-1}\}$ and $\rightarrow_\sigma \supseteq \{(e_i, e_j) : R_i \not\parallel R_j\}$.*

We write $PO(\mathscr{A})$ for the set of section schedules of $\mathscr{A}$. Given a node $v$ in an ART that admits fairness and a section path $\pi$ that starts at $v$, we write $\sigma(\pi)$ for the (unique) section schedule of $\pi$.

While section schedules represent scheduling constraints for execution fragments, we obtain scheduling constraints for complete executions by connecting several section schedules into a *program schedule*. In order to guarantee that each

execution that corresponds to a path in an ART $\mathscr{A}$ adheres to the scheduling constraints of a program schedule $\Sigma$ for $\mathscr{A}$, we require that $\Sigma$ contains a section schedule for at least the initial node and all branching nodes of $\mathscr{A}$.

**Definition 36 (program schedule).** *Given an ART $\mathscr{A}$ that admits fairness with root $\epsilon$, the* program schedule of $\mathscr{A}$ *is a labeled graph $\Sigma = (V_\Sigma, \rightarrow_\Sigma)$ such that:*

- $V_\Sigma \subseteq V_\mathscr{A}$ *($\Sigma$'s nodes are a subset of $\mathscr{A}$'s nodes)*
- $\epsilon \in V_\Sigma \wedge \{v \in V_\mathscr{A} : branching(v)\} \subseteq V_\Sigma$ *($\Sigma$ contains $\mathscr{A}$'s initial node and all branching nodes)*
- $\rightarrow_\Sigma \subseteq V_\Sigma \times PO(\mathscr{A}) \times \mathscr{T} \times \mathscr{F}(Q) \times V_\Sigma$ *(edges are labeled with a section schedule, a thread, and a transition)*
- $\forall v \in V_\Sigma . \exists T \in \mathscr{T} . \forall R \in Transitions(\mathfrak{l}_T(v)) . \exists u \in V_\mathscr{A} . v \xrightarrow{T,R}_\mathscr{A} u \wedge \exists w \in V_\Sigma . \exists \sigma \in PO(\mathscr{A}) . v \xrightarrow{\sigma,T,R}_\Sigma w$ *(every node $v$ has an outgoing edge for each transition of a thread $T$ expanded at $v$ in $\mathscr{A}$ and $T$ is fully expanded in $\mathscr{A}$)*
- $\forall (v, \sigma, T, R, w) \in \rightarrow_\Sigma . \exists \pi . \pi = v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n \wedge v \xrightarrow{\pi}_\mathscr{A} w \wedge \sigma(\pi) = \sigma \wedge T = T_0 \wedge R = R_0$ *(every edge corresponds to a section path in $\mathscr{A}$ that starts with the thread and transition of the edge)*

Similar to schedulers induced by IVRs, a scheduler can enforce the scheduling constraints of a program schedule by looking up a section schedule that matches the current execution prefix and scheduling an event that has only already-executed predecessors, according to the section schedule. All events of a section schedule have to appear before the first event of the next section schedule is executed, so that the states reached between sections correspond to nodes of the program schedule. We formalize this requirement as follows.

**Definition 37 (execution schedule).** *Given a (possibly infinite) path $v_1 \xrightarrow{\sigma_1,T_1,R_1} v_2 \xrightarrow{\sigma_2,T_2,R_2} \ldots$ with $\sigma_i = (V_i, \rightarrow_i), i \geq 1$ in a program schedule, we define its* execution schedule *as the partial order $(V_1 \uplus V_2 \uplus \ldots, (\rightarrow_1 \uplus \rightarrow_2 \uplus \ldots) \cup V_1 \times V_2 \cup V_2 \times V_3 \cup \ldots)$, where $\uplus$ denotes a disjoint union.*

An execution $\tau$ adheres to the scheduling constraints of a program schedule $\Sigma$ if $\tau$ is a linear extension of the execution schedule of some path in $\Sigma$.

**Definition 38 (semantics of program schedules).** *The semantics of a program schedule $\Sigma$ is defined as the set of all executions that are a linear extension of an execution schedule of a path in $\Sigma$.*

**Lemma 9 (correctness of section schedules).** *Let $\tau$ be a linear extension of a section schedule $\sigma(\pi)$ of a section path $\pi$ in a deadlock-free ART $\mathscr{A}$. $\tau$ is equivalent to a linear extension of $\sigma(\pi)$ that corresponds to $\pi$.*

*Proof. Let $\pi$ be a section path, $\sigma(\pi)$ its section schedule, and $\tau$ a linear extension of $\sigma(\pi)$. As $\sigma(\pi)$ is a partial order, all linear extensions of $\sigma(\pi)$ are equivalent [17], in particular the linear extension of $\sigma(\pi)$ that corresponds to $\pi$.*

**Lemma 10 (correctness of program schedules).** *Let $\mathscr{A}$ be an ART that admits fairness and $\Sigma$ a program schedule for $\mathscr{A}$. All program executions that adhere to the scheduling constraints of $\Sigma$ are equivalent to an execution that corresponds to a path in $\mathscr{A}$.*

*Proof. Let $\mathscr{A}$ be an ART that admits fairness, $\Sigma$ a program schedule for $\mathscr{A}$, and $\tau$ be an execution that adheres to the scheduling constraints of $\Sigma$. We show that all finite prefixes $\tau'$ of $\tau$ are equivalent to an execution prefix that corresponds to a path from $\epsilon$ in $\mathscr{A}$.*

*Induction on the length of $\tau'$.*

**case $\tau'$ is empty:** *$\tau'$ corresponds to the empty path in $\mathscr{A}$.*

**inductive case:** *Let $\pi_{\tau'} = v_0 \xrightarrow{\sigma_0(\pi_0)}_\Sigma \dots v_n \xrightarrow{\sigma_n(\pi_n)}_\Sigma v_{n+1}$ be the path in $\Sigma$ that $\tau'$ corresponds to. Let $\tau' = x_1 x_2$ be partitioned so that $x_1$ corresponds to the prefix $v_0 \dots v_n$ in that path. Such a partition exists, as by Definition 37, an event must occur after all events from the previous section schedule and before all events from the following section schedule.*
*By induction hypothesis, there exists an execution $x_1^{\approx}$ that is equivalent to $x_1$ that corresponds to the path $\pi_0 \dots \pi_{n-1}$ in $\mathscr{A}$. By Lemma 9, there exists a linear extension $x_2^{\approx}$ of $\sigma_n(\pi_n)$ that is equivalent to $x_2$, which corresponds to $\pi_n$ in $\mathscr{A}$. Thus, $x_1^{\approx} x_2^{\approx}$ is equivalent to $\tau'$ and corresponds to $\pi_0 \dots \pi_n$.*

Algorithm 3 shows how an IVR can be derived from a program schedule. The algorithm requires a program schedule for the program under execution and maintains a current section schedule $\sigma$. Given an execution prefix $\tau$, it checks whether there are still events in $\sigma$ that are not yet executed. If this is not the case, the current section is reset to a section that is feasible in the current state. Afterwards, those events that have already been executed are temporarily removed from $\sigma$ and a thread is scheduled that has no predecessors in $\sigma$ after this removal.

**Algorithm 3:** IVR induced by a program schedule $\Sigma$

**input :** $s_0 T_0 \dots s_n$, an execution prefix
**output:** a set of threads that is permitted to execute after the given execution prefix
**Data:** $\Sigma$, a program schedule
**Data:** $\sigma$, initially some section schedule such that $\epsilon \xrightarrow{\sigma}_\Sigma w$ for some $w$
**Data:** $v := w$
**Data:** $i := 0$
**Function** $R(s_0 T_0 \dots s_n)$
  **if** $n - i = |\sigma|$ **then**
    $i := n$
    choose $\sigma, w$ s.t. $v \xrightarrow{\sigma,T,R}_\Sigma w$ and $s_n$ satisfies the guard of transition $R$
    $v := w$
  $\sigma' := \sigma$ with the events of $T_i, \dots, T_{n-1}$ removed
  **return** $\min(\sigma')$ *(all threads that have no predecessors in $\sigma$)*

## C Iterative model checking

**Lemma 11 (fairness of iterative Impact).** *Whenever Alg. 1 yields a safe ART $\mathscr{A}$, $\mathscr{A}$ is fair.*

*Proof. By contradiction. Assume that Alg. 1 returns a safe ART* $\mathscr{A} = (V_{\mathscr{A}}, \epsilon, \rightarrow_{\mathscr{A}}, \triangleright)$ *that is not fair. By definition 31,* $\mathscr{A}$ *contains a* $(\triangleright \cup \rightarrow_{\mathscr{A}})$*-cycle* $c$ *that does not satisfy fair*$(c)$*. As* $(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ *is a tree, the cycle contains a* $\triangleright$ *edge. However, Alg. 1 checks, in line 11, whether the candidate covering would produce an unfair cycle. A* $\triangleright$ *edge is only added if the resulting cycle is fair. Contradiction.*

## D Additional experimental results

Table 2: Additional experimental results (to: timeout)

| | Model checking | | Performance (higher is better) | | | |
|---|---|---|---|---|---|---|
| Benchmark | Time 1st IVR | Impara-C | Opt0 | Opt1 | Opt2 | Unconstrained |
| prod.-cons. 1p 1c 1000b | **2 m 0 s** | to (72h) | 4 864 489 | 7 466 093 | **11 370 258** | 8 199 202 |
| prod.-cons. 2p 2c 1000b | **23 m 47 s** | to (72h) | 3 400 187 | 5 959 041 | 8 428 598 | **11 643 208** |
| prod.-cons. 4p 4c 1000b | **4 h 29 m 53 s** | to (72h) | 1 327 063 | 2 576 695 | 3 676 876 | **7 210 796** |
| prod.-cons. 1p 1c 5b | **2 s** | 2 m 28 s | 4 945 116 | 7 075 596 | **12 372 817** | 7 915 465 |
| prod.-cons. 2p 2c 5b | **18 s** | 1 m 16 s | 3 194 019 | 5 514 429 | **9 271 859** | 6 933 172 |
| prod.-cons. 4p 4c 5b | **2 m 41 s** | 9 m 44 s | 1 345 991 | 2 465 108 | **3 392 111** | 3 240 136 |
| double lock 1 ms | 0 s | 0 s | 1 845 | 1 834 | **3 217** | 1 797 |
| file system | 0 s | 0 s | 3 667 | 4 877 035 | 6 705 672 | **23 822 129** |
| barrier | **1 s** | 4 m 14 s | 1 238 720 | 8 285 228 | **14 586 849** | 1 077 907 |

Tab. 2 contains additional experimental result for the producer-consumer benchmark with a buffer size of 5 (5b) instead of 1000 (1000b). As the first IVR does not make use of more than at most four cells in the buffer (in case of four producers), the effect on Opt0–Opt2 is small, as expected. However, the performance of unconstrained executions decreases with a smaller buffer.

| | Execution time (s) | | |
|---|---|---|---|
| Schedule | Constrained | Unconstrained | Relative |
| S1 | 3.34 | 3.25 | 1.03 |
| S2 | 3.34 | 3.25 | 1.03 |
| S3 | 3.6 | 3.25 | 1.10 |
| S4 | 3.57 | 3.25 | 1.10 |

Table 3: Experimental performance results for pfscan

Tab. 3 contains our experimental results for the pfscan benchmark. We use two worker threads in addition to the main thread. The benchmark is executed with scheduling constraints of several program schedules S1–4 (column two) and unconstrained (column three). Execution times are given in seconds. The fourth column gives the relative execution time (overhead). In all constrained configurations, operations on synchronization objects have been omitted (Opt1). S1, S2, and S3 are program schedules as they can be produced during the first iteration of our model checking algorithm. Program schedule S4 allows any interleaving of critical sections so that all executions of the unconstrained program are matched. S1 and S2 contain sections that comprise both worker threads, while S3 and S4 contain only single-threaded sections. S1 and S2 differ in the ordering of the worker threads.

S3 causes an overhead of 10% with respect to the unconstrained execution. Although S4 allows any interleaving of critical sections, there remains an overhead of 10% caused by looking up section schedules during the execution. S1 and S2 show only a small overhead of 3%. We conjecture that the lower number of section schedule look-ups (compared to S3 and S4) is responsible for the considerably lower overhead.

## E Enabled threads

An ART $\mathscr{A}$ may contain an edge $v \xrightarrow{T,a} w$ such that transition $a$ is enabled in some state $s \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$ but disabled in some state $s' \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$. This may pose a problem when the goal is to construct an ART that admits fairness, as Definition 30 requires an edge $v \xrightarrow{T,a} w$ that is enabled in all states that correspond to $v$, for every cycle and thread that is enabled in that cycle. We argue that the situation above cannot occur, even when constructing an ART with the conventional Impact algorithm for concurrent programs [40], if programs make only "reasonable" use of locks, as described below.

We restrict programs such that whether an transition is enabled in a state $s$ may only depend on the global location $\mathfrak{l}()(s)$ of s but not on the variable valuation of $s$. Formally:

$$\forall T \in \mathscr{T}.\, \forall a \in R_T.\, \exists f : (\mathscr{T} \to L) \to \{0,1\}.\, \forall s.\\ (a \in \mathit{enabled}(s) \Leftrightarrow f(\mathfrak{l}()(s)) = 1) \quad (1)$$

For such programs, every transition $a$ with an edge $v \xrightarrow{T,a} w$ in a well-labeled ART is trivially enabled in all states $s \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$. In the following, we argue that such programs are sufficient to express "reasonable" uses of locks.

We assume that there exists a synchronization primitive lock(l) that acquires the lock l if it is free and otherwise lets the executing thread wait until l is free. A thread is *disabled* when its next statement is lock(l) for a lock l that is not free. Furthermore, we assume that lock is the only primitive in the targeted programming language that can disable threads (other synchronization constructs can be built using lock).

Consider a program $P$ that, for every lock statement stmt that occurs in $P$, always executes stmt with the same lock. $P$ satisfies (1). On the other hand, consider a program $P'$ that maintains an array of locks locks = [l_1, l_2, ..., l_n] and contains a statement lock(locks[*]) that tries to acquire a non-deterministically chosen lock. $P'$ does not satisfy (1).

A pattern that violates (1) may be translated so that a unique lock is used at a given program location as follows. A program fragment (where l is a local variable)

```
1  l = locks[*];
2  lock(l);
3  critical_section();
4  unlock(l);
```

is translated to:

```
1  l = locks[*];
2  switch l:
3    case l_1:
4      lock(l_1);
5      critical_section_1();
6      unlock(l_1);
7    case l_2:
8      lock(l_2);
9      critical_section_2();
10     unlock(l_2);
11   ...
12   case l_n:
13     lock(l_n);
14     critical_section_n();
15     unlock(l_n);
```

This transformation leads to a linear blow up in program size. However, we assume that practical programs which violate (1) are rare and call programs on which above transformation does not critically increase program size *programs with a "reasonable" use of locks*. For such programs, an ART that is an incomplete product of the conventional Impact algorithm for concurrent programs can be easily extended to an ART that admits fairness.

## F Auxiliary lemmas

The following lemma is used in the proofs of Lemma 7 and 8. It states that for every node $v$ of a finite graph that is visited infinitely often in a path, this path also visits infinitely often all nodes of a cycle that contains $v$.

**Lemma 12 (completely visited cycles).** *Let $G = (V, \to)$ be a directed, finite graph. For all infinite paths $\pi \in V^\omega$ through $G$ and for all nodes $v \in V$ that occur infinitely often in $\pi$, there exists a projection $\pi' \subseteq \pi$ such that $\pi' = \pi_v^\omega$ and $\pi_v$ is a cycle that contains $v$, i.e., there exists a cycle $\pi_v$ in $G$ that contains $v$ such that by removing nodes from $\pi$, we obtain a path $\pi' = \pi_v^\omega$ that visits all nodes of $\pi_v$ infinitely often.*

*Proof. Let $G, \pi, v$ be as in the lemma. $\pi$ has the form $\pi_0 \circ v \circ \pi_1 \circ v \circ \pi_2 \circ v \cdots$ with $v \notin \pi_i, i \geq 0$. For all $i \geq 1$, $v \circ \pi_i \circ v$ is a closed walk, which can be shortened to a cycle $v \circ \pi_i' \circ v$. As there are only finitely many cycles in $G$ ($V$ is finite), there exists a cycle $v \circ \pi_{i_1}' \circ v$ that is repeated infinitely often in the sequence $\pi_0 \circ v \circ \pi_1' \circ v \circ \pi_2' \circ v \cdots$, i.e., $\pi_{i_1}' = \pi_{i_2}' = \pi_{i_3}' = \cdots$ for an infinite sequence of indices $i_1 < i_2 < i_3 < \cdots$. Let $\pi_v = v \circ \pi_{i_1}$. We have that $\pi' = \pi_v^\omega$ is a projection of $\pi$ and $\pi_v$ is a cycle that contains $v$.*

## G Transformation of fair ARTs to independently fair ARTs

Given a fair, well-labeled, safe ART $\mathscr{A}$, Algorithm 4 generates an independently fair ART $\mathscr{A}'$ such that $\mathit{Executions}(\mathcal{R}_{\mathscr{A}'}) \subseteq \mathit{Executions}(\mathcal{R}_{\mathscr{A}})$.

**Algorithm 4:** Transformation of fair ARTs to independently fair ARTs

**input :** fair ART $\mathscr{A}$
**output:** independently-fair ART $\mathscr{A}'$ with $Executions(\mathcal{R}_{\mathscr{A}'}) \subseteq Executions(\mathcal{R}_{\mathscr{A}})$
**Data:** $\mathscr{A}' := \emptyset$, $W := \epsilon$
**while** $\exists v \in W$ **do**
  remove $v$ from $W$;
  **if** *$v$ can be independently-fair covered by some node $w \in \mathscr{A}'$* **then**
    add $\{v \rhd w\}$ to $\mathscr{A}'$;
    continue;
  **else if** *$v$ is part of a $(\rhd_{\mathscr{A}} \cup \rightarrow_{\mathscr{A}})$-cycle $v \ldots wv$ that is not independently fair* **then**
    add $\{v \rightarrow \ldots \rightarrow w\}$ as fresh nodes to $\mathscr{A}'$;
    set $v$ to an exit node of the cycle that has not yet been expanded;
    add $\{w \rightarrow \ldots \rightarrow v\}$ to $\mathscr{A}'$;
  add $\{v \rightarrow v' : v \rightarrow_{\mathscr{A}} v'\}$ to $\mathscr{A}'$;
  add $\{v \rhd v' : v \rhd_{\mathscr{A}} v'\}$ to $\mathscr{A}'$;
  add $\{v' : v \rightarrow_{\mathscr{A}} v' \vee v \rhd_{\mathscr{A}} v'\}$ to $W$;

# H Iterative Impact for concurrent programs

**Algorithm 5:** Iterative Impact for concurrent programs

**Function** *Refine(v)*
  **if** $\mathfrak{l}(v) \neq \mathfrak{l}_{error}$ *or* $\phi(v) \equiv false$ **then**
    **return**
  $\pi := v_0, \ldots, v_n$ path from $\epsilon$ to $v$
  **if** *path formula of* $\pi$ *has interpolant* $A_0, \ldots, A_n$ **then**
    **for** $i = 0 \ldots n$ **do**
      $\phi := A_i^{-i}$
      **if** $\phi(v_i) \not\models \phi$ **then**
        $W := W \cup \{w : w \rhd v_i\}$
        $\rhd := \rhd \setminus \{(w, v_i) : w \rhd v_i\}$
        $\phi(v_i) := \phi(v_i) \wedge \phi$
    **for** $w \in V$ *such that* $v$ *is a descendant of* $w$ **do**
      `Close` $(w)$
  **else**
    **return** *counterexample*

**Function** *Expand_Thread(T, v)*
  **for** $R_{\mathfrak{l},\mathfrak{l}'} \in$ *Transitions*$(\mathfrak{l}_T(v))$ **do**
    $w :=$ fresh node
    $\mathfrak{l}(w) := \mathfrak{l}(v)[T \mapsto \mathfrak{l}']$
    $\phi(w) :=$ True
    $W := W \cup \{w\}$
    $V := V \cup \{w\}$
    $\rightarrow := \rightarrow \cup \{(v, T, R, w)\}$

**Function** *Skip(v, T)*
  **if** $(v, T) \in I$ **then**
    **return** *false*
  **else**
    choose unique $T', a'$ such that $u \xrightarrow{T', a'} v$
    **return** $(T < T' \wedge (Transitions(v)T \parallel \{a'\})) \wedge \neg Loop(u, T')$

**Function** `Schedule_Thread` *(v)*
  let $R_n$ be the transition of thread $T_n$ by which $v$ is reached
  **if** $R_n$ *represents a back jump* **then**
    $T := T_n + 1 \mod |\mathscr{T}|$
  **else**
    $T := T_n$
  **while** `Skip` *(v, T)* **do**
    $T := T + 1 \mod |\mathscr{T}|$
  **return** $T$

In order to represent a path of length $n$ as a formula, we define $n$ fresh copies of the set of variable symbols, denoted by $Q_1, \ldots, Q_n$, such that $Q_1$ is equal to the previously defined copy $Q'$ for transition formulae. The *path formula* of a path $\pi = v_0 \xrightarrow{T_0, R_0}_{\mathscr{A}} \ldots \xrightarrow{T_{n-1}, R_{n-1}}_{\mathscr{A}} v_n$ in an ART $\mathscr{A}$ is the formula $\phi(v_0) \wedge R_0 \wedge R_1^1 \wedge \cdots \wedge R_{n-1}^{n-1} \in \mathscr{F}(Q \cup Q' \cup Q_1 \cup \cdots \cup Q_n)$, where $R_i^i, 1 \leq i \leq (n-1)$ is obtained from $R_i \in \mathscr{F}(Q \cup Q')$ by substituting the variable symbols in $Q$ and $Q'$ with their corresponding copies of $Q_i$ and $Q_{i+1}$, respectively. We write $A^{-i}$ for some formula $A$ to reverse this substitution, i.e., $A = (A^i)^{-i}$. This notation is used in Alg. 5 to construct a sequent interpolant $A_0, \ldots, A_n \in \mathscr{F}(Q \cup Q' \cup Q_1 \cup \cdots \cup Q_n)$ for a path formula and extract state formulas $A_i^{-i} \in \mathscr{F}(Q)$.

We use the set $I$ to record cases in which POR would hide a transition $w \xrightarrow{T} w'$ after adding a covering $(v, w) \in$ *covering*(). The original approach

by Wachter et al. [40] of immediately expanding a thread after adding such a covering does not suite our iterative variant of the algorithm, as this approach could explore more than one schedule in a single iteration. Instead of immediately exploring $w \xrightarrow{T} w'$, we record this transition in $I$ and prevent the procedure *Skip()* from skipping it (i.e., applying POR).