**Noname manuscript No.**
(will be inserted by the editor)

# Extracting Safe Thread Schedules from Incomplete Model Checking Results

**Patrick Metzler · Neeraj Suri · Georg Weissenbacher**

the date of receipt and acceptance should be inserted later

**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.

P. Metzler supported by the German Academic Exchange Service (DAAD).

N. Suri supported in part by H2020-SU-ICT-2018-2 CONCORDIA GA #830927 and BMBF-Hessen TUD CRISP.

G. Weissenbacher funded by the Vienna Science and Technology Fund (WWTF) through the project Heisenbugs (VRG11-005) and the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF).

P. Metzler
TU Darmstadt
E-mail: patrick.metzler@posteo.net

N. Suri
Lancaster University
E-mail: neeraj.suri@lancatser.ac.uk

G. Weissenbacher
TU Wien
E-mail: georg.weissenbacher@tuwien.ac.at

## 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. Our contribution is the novel generation and use of IVRs based on existing model checking algorithms, where we use lazy abstraction with interpolants [40] to instantiate our approach. The scheduling constraints induced by an IVR can be enforced by *iteratively relaxed scheduling* [29], a technique to enforce fine-grained orderings of concurrent memory events. When the scheduling constraints of an IVR are enforced, all executions (for all possible inputs) are safe, even if the underlying (operating system) scheduler is non-deterministic. Thereby, the program can be executed safely before a 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 syn-

```
 1 initially:
 2   empty buffer of size N
 3   count = 0
 4   mutex = 0
 5
 6 thread $T_1$:
 7   while true:
 8     produce()
 9
10 thread $T_2$:
11   while true:
12     consume()
13 produce:
14   lock(mutex)
15   if count < N:
16     put item
17     count += 1
18   else:
19     error (overflow)
20   unlock(mutex)
21
22 consume:
23   lock(mutex)
24   if count > 0:
25     remove item
26     count −= 1
27   else:
28     error (underflow)
29   unlock(mutex)
```

Fig. 1: An erroneous version of the producer-consumer problem

chronization 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 producer causes an overflow or the consumer causes 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 IVRs, the final result would be *safe* or *unsafe*, respectively.

This paper shows how to instantiate our approach by answering the following 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.

Beyond the contributions of a previous version of this paper [30], this extended version contains proofs of our formal statements, a more detailed description of constructing ARTs with the monolithic Impact algorithm for concurrent programs and our iterative extension, a more detailed description of the implementation for our evaluation, additional experimental performance measurements, additional illustration of our case studies, and a more detailed discussion of section schedules and their optimization.

## 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 15 to 16 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 $s$ satisfies $\mathit{Guard}(R)$. We write $\mathit{enabled}(s)$ for the set of enabled transitions at $s$. 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. If $R$ exists, we write $\mathit{enabled}_T(s) := \{R\}$ and $\mathit{enabled}_T(s) := \emptyset$ otherwise.

If there exist states $s$ for which no transition of a thread $T$ is enabled (e.g., in line 14 in Fig. 1), $T$ may block. We assume that such locations $\mathfrak{l}_T(s)$ are (conservatively) marked by $\mathit{may\text{-}block}(\mathfrak{l}_T(s))$.

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 and enforce it by our algorithm presented in Sec. 3. Other notions of fairness, such as weak fairness, can be enforced analogously to our use of strong fairness.

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 $R_2$. If $R_1$ and $R_2$ are not independent, we write $R_1 \nparallel R_2$.

## 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, 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 $\zeta_{\mathcal{R}}$ *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 ARTs (left) and IVRs (right).

*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.

| ART: | | IVR: |
|---|---|---|
| $\mathscr{A}$ is safe | $\Rightarrow$ | $\mathcal{R}_{\mathscr{A}}$ is safe |
| | | $\Uparrow$ |
| $\Uparrow$ | | $\mathcal{R}_{\mathscr{A}}$ is realizable |
| | | $\Uparrow$ |
| $\mathscr{A}$ is deadlock-free | $\Rightarrow$ | $\mathcal{R}_{\mathscr{A}}$ is deadlock-free |
| $\Uparrow$ | | $\Uparrow$ |
| $\mathscr{A}$ admits fairness | $\Rightarrow$ | $\mathcal{R}_{\mathscr{A}}$ admits fairness |
| $\Uparrow$ | | $\Uparrow$ |
| $\mathscr{A}$ is fair | $\Rightarrow$ | $\mathcal{R}_{\mathscr{A}}$ is fair |

Fig. 2: Overview on the relationship between properties of IVRs and ARTs. $\Rightarrow$ and $\Uparrow$ denote logical implication.

### 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 [21,28,23,9] 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 pathwise 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 ($\mathit{covered}(v)$) 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 [28,40])** 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 $v \xrightarrow{T,R} w$.

Intuitively, an ART $\mathscr{A}$ is *well-labeled* [28] if $\mathscr{A}$'s $\rightarrow$-edges represent the transitions of the program and edges $v \rhd w$ indicate that all states modeled by node $v$ are also modeled by node $w$. Formally, $\mathscr{A}$ is well-labeled if for every edge $v \xrightarrow{T,R_{\mathfrak{l},\mathfrak{l}'}} w$ in $\mathscr{A}$ we have that (i) $\phi(\epsilon)$ represents the 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 $\neg\mathit{covered}(w)$.

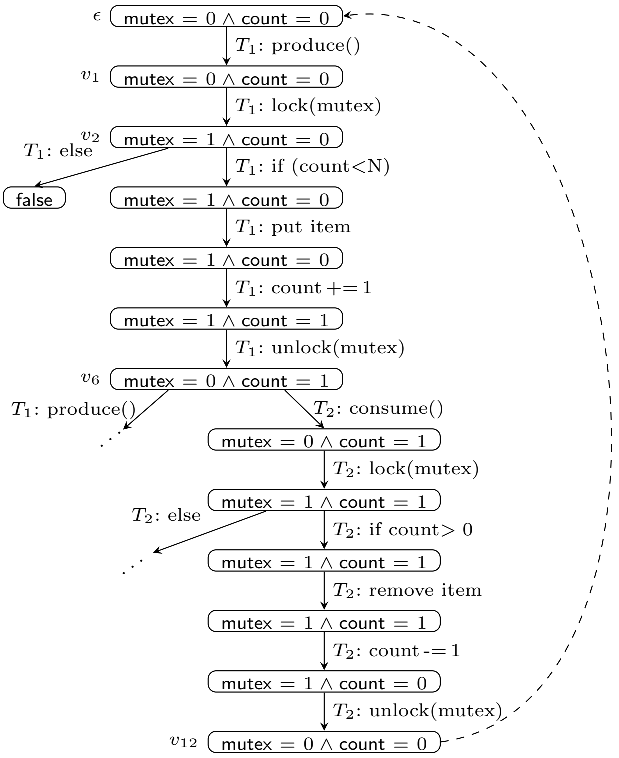


Fig. 3: An (incomplete) ART for the program of Fig. 1

An incomplete ART $\mathscr{A}_{\text{p-c}}$ for the producer-consumer problem of Fig. 1 is shown in Fig. 3. Nodes show the state formulas and edges are labeled with the thread and statement corresponding to the transition. The dashed edge is a $\rhd$-edge.

*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 $\tau = s_0, T_1, s_1, \dots, s_n$ 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 $w$). Execution prefixes are matched with ($\rhd \cup \rightarrow$)-paths, which is, in particular, necessary to build infinite executions. For example, the execution prefix

$$\tau = s_0,\ \underbrace{T_1, s_1, \dots, T_1,}_{T_1 \text{ scheduled 6 times}}\ s_6,\ \underbrace{T_2, s_7, \dots, T_2,}_{T_2 \text{ scheduled 6 times}}\ s_0$$

corresponds to the path in $\mathscr{A}_{\text{p-c}}$ from $\epsilon$ over $v_1, \dots, v_{12}$ back to $\epsilon$. As only $T_1$ is expanded at $\epsilon$, $\mathcal{R}_{\mathscr{A}\text{p-c}}$ allows only $\{T_1\}$ after $\tau$.

*Safety.* An ART is *safe* if whenever $\mathfrak{l}_T(v)$ is the error location then $\phi(v) = \mathit{false}$. 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 transition of $T$ is enabled (i.e., $\mathit{enabled}_T(s) = \emptyset$). 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 $\mathit{may\text{-}block}(\mathfrak{l}_T)$).

Consequently, whenever $\mathit{may\text{-}block}(\mathfrak{l}_T(v))$ in a *deadlock-free* ART $\mathscr{A}$, we require that $\phi(v)$ is strong enough to entail that the transition $R$ of $T$ expanded at $v$ (or at the node covering $v$, respectively) are enabled (i.e., $\phi(v) \Rightarrow \mathit{Guard}(R)$). 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.*

*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.*

*For arbitrary execution prefixes of the form $\tau = s_0, T_1, s_1, \ldots, s_n$, let $\mathscr{T}'(\tau) = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T \in \mathscr{T} : \mathit{enabled}_T(s_n) \neq \emptyset\}$. 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 $\zeta(\tau) = T$, $\mathit{enabled}_T(s_n) = \emptyset$ 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 $\mathit{enabled}_T(s_n) = \emptyset$.*

*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 $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$: 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 the assumption that $\tau$ corresponds to a path $\pi$, $s_n \vDash \phi(v_n)$. Hence, $\phi(v_n) \vDash \mathit{Guard}(R)$ and $R \in \mathit{enabled}(s_n)$. Contradiction to $\mathit{enabled}(s_n) = \emptyset$.*

*case not $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$: By the definition of* may block*, $\mathit{enabled}_T(s_n) \neq \emptyset$. Contradiction to $\mathit{enabled}_T(s_n) = \emptyset$.*

*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 $\tau = s_0, T_1, s_1, \ldots, s_n$ that is possible under $\zeta$ such that $s_n$ is a deadlock, i.e., $\forall T \in \mathscr{T}. \mathit{enabled}_T(s_n) = \emptyset$ and $\exists T \in \mathscr{T}. \exists R_{\mathfrak{l},\mathfrak{l}'}. \mathfrak{l}_T(s_n) = \mathfrak{l}$. As $\tau$ is an execution permitted by $\mathcal{R}_{\mathscr{A}}$, $\tau$ 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 $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$, we have $\phi(v_n) \Rightarrow \mathit{Guard}(R)$ for some transition $R_{\mathfrak{l},\mathfrak{l}'}$ with $\mathfrak{l}_T(v_n) = \mathfrak{l}_T(s_n) = \mathfrak{l}$ and a contradiction to $\mathit{enabled}(s_n) = \emptyset$ and in case not $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$, we have $\mathit{enabled}_T(s_n) \neq \emptyset$ and a contradiction to $\mathit{enabled}_T(s_n) = \emptyset$.*

*Fairness.* IVRs derived from deadlock-free ARTs do not necessarily admit fairness if the underlying ART contains cycles (across $\rhd$ and $\to$ 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 $(\rhd \cup \to)$-cycle is "fair", defined as follows.

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

Before we proof the fairness of IVRs induced by fair ARTs, we state the following auxiliary proposition.

**Proposition 1 (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 cycle $\pi'$ in $G$ such that $\pi'$ contains $v$ and all nodes of $\pi'$ are visited infinitely often by $\pi$.*

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

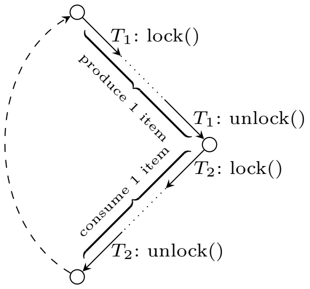


Fig. 4: A $(\rhd \cup \rightarrow)$-cycle ($\rhd$ is shown by a dashed line)

*$\pi = v_0, T_1, \ldots, v_n$. 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., $\mathscr{T}' = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T : enabled_T(s_n) \neq \emptyset\}$. 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 1 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 Proposition 1, 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 $\pi$.*

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 $(\rhd \cup \rightarrow)$-cycle $c$, which we denote by $fair(c)$. The $(\rhd \cup \rightarrow)$-cycle shown in Fig. 4, for instance, is fair.

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

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

*Proof Let $\mathscr{A}$ be a fair ART. By Lemma 1 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. As $\mathscr{A}$ is finite and by Proposition 1, $v$ 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 $\mathcal{R}_{\mathscr{A}}$ permits all executions permitted by $\mathcal{R}_{\mathscr{A}'}$.

## 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 techniques such as symbolic execution are also suitable to generate IVRs. In particular, our algorithm (Alg. 1) constitutes an iterative extension of the Impact algorithm [28] 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, the node labeling is strengthened via interpolation. Thereby, a well-labeled ART is maintained. Once the ART is complete, its node labeling provides a safety proof for the program.

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

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

Function Main()
    while $true$ do
        status := Iteration()
        if $status$ = $no\ progress$ then
            break
        else if $status$ = $counterexample$ then
            yield $\mathscr{A}$ as an unsafe IVR
        else
            $\mathscr{A}'$ := Remove_Error_Paths($\mathscr{A}$)
            yield $\mathscr{A}'$ as a safe IVR
    if $\mathscr{A}$ is safe then
        return $safe$
    else if Remove_Error_Paths($\mathscr{A}$) admits fairness then
        return $partially\text{-}safe$
    else
        return $unsafe$

Function Iteration()
    $W$ := New_Schedule_Start()
    if $W = \emptyset$ then
        return $no\ progress$
    while $W \neq \emptyset$ do
        select and remove $v$ from $W$
        Close($v$)
        if $v$ not covered then
            status := Refine ($v$)
            if $status$ = $counterexample$ then
                return $counterexample$
            status := Check_Enabledness($v$)
            if $status$ = $no\ progress$ then
                return $no\ progress$
            Expand ($v$)
    return $progress$

Function Check_Enabledness($v$)
    $\pi := v_0 \xrightarrow{T_1, R_1} v_1 \ldots \xrightarrow{T_n, R_n} v_n$ path from $\epsilon$ to $v$
    if not may-block($\mathfrak{l}v_{n-1}$)$T\_n$ then
        return $progress$
    if $R_1 \wedge \ldots \wedge R_{n-1} \wedge \neg Guard(R_n)$ is unsat then
        $\phi(v) := \phi(v) \wedge Guard(R_n)$
    else
        return Backtrack($v$)

Function Close($v$)
    for all uncovered nodes $w$ that have been created before $v$ do
        if $\mathfrak{l}(w) = \mathfrak{l}(v) \wedge (\phi(v) \Rightarrow \phi(w)) \wedge \forall c \in C_{\mathscr{A}}(v, w).\ fair(c)$ then
            $\rhd := \rhd \cup \{(v, w)\}$
            $\rhd := \rhd \setminus \{(x, y) : v \rightsquigarrow y\}$
            for $T$ with $v \xrightarrow{T} v'$ and not $w \xrightarrow{T} w'$ do
                add $(v, T)$ to $I$

Function Backtrack($v$)
    $\pi := v_0 \xrightarrow{T_1, R_1} v_1 \ldots \xrightarrow{T_n, R_n} v_n$ path from $\epsilon$ to $v$
    $i := n - 1$
    while $i \geq 0$ do
        if $\exists T, v_i'.\ v_i \xrightarrow{T} v_i' \notin \mathscr{A}$ $\wedge$(Skip($v_i$, $T$) = $false$) then
            add $v_i \xrightarrow{T} v_i'$ to $\mathscr{A}$
            $W := W \cup \{v_i'\}$
            prune $\xrightarrow{T_{i+2}, R_{i+2}} v_{i+3} \ldots\ldots \xrightarrow{T_n, R_n} v_n$ from $\mathscr{A}$
            $\phi(v_{i+1}) := false$
            return $progress$
        $i := i - 1$
    return $no\ progress$

Function Expand($v$)
    $T$ := Schedule_Thread ($v$)
    Expand_Thread ($T$, $v$)
```

To build an ART as in the producer-consumer example of Fig. 3, IMPACT starts by constructing the root node $\epsilon$ with $\phi(\epsilon)$ = true and $\mathfrak{l}(\epsilon) = (8, 12)$, where we indicate locations by line numbers in Fig. 1. Initially, $\mathsf{mutex} = 0$, $\mathsf{count} = 0$, and the buffer size is bound by an arbitrary constant $\mathsf{N} > 0$. Thread $T_1$ is expanded by adding a node $v_1$ with $\phi(v_1)$ = true and $\mathfrak{l}(v_1) = (14, 12)$. From $v_1$, thread $T_1$ is expanded repeatedly until node $v_6$ with $\phi(v_6)$ = true and $\mathfrak{l}(v_6) = (8, 12)$ is produced. At this point, all statements of the $\mathsf{produce()}$ procedure have been expanded once. As $v_6$ has the same global location as $\epsilon$ and $\phi(v_6) \Rightarrow \phi(\epsilon)$, a covering $v_6 \rhd \epsilon$ can be inserted. However, when the else branch of thread $T_1$ at node $v_1$ is expanded, a node $v_{\text{error}}$ labeled with the error location is added. In order to check the feasibility of the error path $\epsilon \rightarrow v_1 \rightarrow v_2 \rightarrow v_{\text{error}}$, IMPACT tries to find a sequence interpolant for:

$$\mathsf{count} = 0 \wedge \mathsf{mutex} = 0,$$

$$\mathsf{mutex}' = 1,$$

$$\mathsf{count} \geq \mathsf{N}$$

As we assume that the buffer is never of size 0, i.e., $\mathsf{N} > 0$, $\bigwedge \mathcal{U}$ is unsatisfiable and a possible sequence interpolant is:

$$I_0 \equiv \text{true}$$
$$I_1 \equiv \mathsf{count} = 0 \wedge \mathsf{mutex} = 0$$
$$I_2 \equiv \mathsf{count} = 0 \wedge \mathsf{mutex}' = 1$$
$$I_3 \equiv \text{false}$$

with:

$$I_0 \wedge \mathsf{count} = 0 \wedge \mathsf{mutex} = 0 \Rightarrow I_1$$
$$I_1 \wedge \mathsf{mutex}' = 1 \Rightarrow I_2$$
$$I_2 \wedge \mathsf{count} \geq \mathsf{N} \Rightarrow I_3$$

Hence, $v_{\text{error}}$ can be labeled with false, so that the ART remains safe, and the preceding labels can be updated to $\phi(\epsilon) = \phi(v_1) = \mathsf{count} = 0 \wedge \mathsf{mutex} = 0$ and $\phi(v_2) = \mathsf{count} = 0 \wedge \mathsf{mutex} = 1$. Due to the relabeling, the covering $v_6 \rhd \epsilon$ has to be removed and $v_6$ has to be expanded.

When $T_2$ has been expanded six times beginning at $v_6$, a node $v_{12}$ is added with $\mathfrak{l}(v_{12}) = (8, 12)$. Impact applies a heuristic that attempts to introduce coverings eagerly, which results in a label $\phi(v_{12}) = \mathsf{mutex} = 0 \wedge \mathsf{count} = 0$ and a covering $v_{12} \rhd \epsilon$ can be added. With this covering, the current ART is fair and can be used as an IVR. In contrast, Impact for concurrent programs would then continue to explore additional interleavings by expanding, e.g., $T_2$ at $\epsilon$. A complete ART is found when both error paths and all interleavings of produce() and consume() that respect the available buffer size N are explored. Impact for concurrent programs does not terminate until such a complete ART is found and would not terminate at all if the buffer size is unbounded. Our algorithm, however, is able to yield an fair IVR each time a new interleaving has been explored.

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 (lines 12, 14, and 16). Otherwise, an intermediate output is yielded: either $\mathscr{A}$ as an intermediate output (line 7) or $\mathscr{A}$ with all previously found counterexamples removed, i.e., the largest fair ART that is a subgraph of $\mathscr{A}$, denoted by *Remove_Error_Paths()*.

*Iteration()* maintains a work list $W$ of nodes $v$ to be explored via *Close(v)*, which tries to find (as in [40]) a node that covers $v$. In addition to the covering check of [40], we check fairness, where $C_{\mathscr{A}}(v, w)$ denotes all cycles that would be closed by adding the edge $v \rhd w$ (line 43). If such a node $w$ is found, any thread $T$ that is expanded at $v$ but not at $w$ (line 46) must not be skipped at $w$ by POR. Instead of expanding $T$ instantaneously at $w$ (as in [40]), which would explore another schedule, $T$ is added to the set $I$ so that it can be explored in a subsequent iteration. If no covering node for $v$ is found, $v$ is refined, which returns *counterexample* if $v$ has a feasible error path (line 25). Otherwise (line 28), *Check_Enabledness()* performs a deadlock check by testing whether the last transition 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); the latter decides which thread to expand at a given node (line 61). 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.

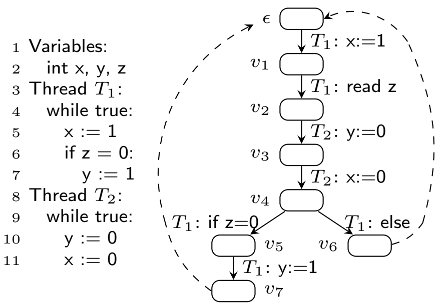
The correctness of Alg. 1 w.r.t. safety follows from the correctness of [28] and [40]. Additionally, Alg. 1 is also fair:

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

*Proof By contradiction. Assume that Alg. 1 returns a safe ART $\mathscr{A} = (V_{\mathscr{A}}, \epsilon, \rightarrow_{\mathscr{A}}, \rhd)$ that is not fair. By definition 5, $\mathscr{A}$ contains a $(\rhd \cup \rightarrow_{\mathscr{A}})$-cycle $c$ that does not satisfy fair$(c)$. As $(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ is a tree, the cycle contains a $\rhd$ edge. However, Alg. 1 checks, in line 43, whether the candidate covering would produce an unfair cycle. A $\rhd$ edge is only added if the resulting cycle is fair. Contradiction.*

## 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



```
Variables:
  int x, y, z
Thread T1:
  while true:
    x := 1
    if z = 0:
      y := 1
Thread T2:
  while true:
    y := 0
    x := 0
```

Fig. 5 (a) A Program with a fair ART

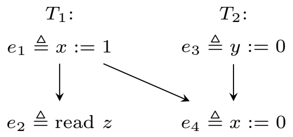


Fig. 5 (b) The section schedule for the section path $\pi_1$ from $\epsilon$ to $v_4$

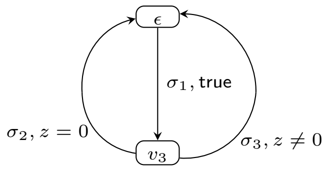


Fig. 5 (c) A corresponding program schedule

introduce *program schedules* that relax the scheduling constraints by means of partial-order reduction (POR). 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. 5a. 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:

$$\pi_1 := \epsilon \to v_1 \to v_2 \to v_3 \to v_4$$
$$\pi_2 := v_4 \to v_5 \to v_7 \to \epsilon$$
$$\pi_3 := v_4 \to v_6 \to \epsilon$$

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* $v_1 \xrightarrow{R_1} \dots \xrightarrow{R_n} v_{n+1}$ corresponds to a branching-free path in an ART whose first transition may be guarded. A section path follows $\to_{\mathscr{A}}$ 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 $\sigma = (V_\sigma, \to_\sigma)$ such that $V_\sigma = \{e_1, \dots, e_n\}$ and $\to_\sigma \supseteq \{(e_i, e_j) : i < j \wedge R_i \not\parallel R_j\}$, where $e_i, 1 \le i \le n$ is the occurrence of transition $R_i$ at position $i$.

The section schedule $\sigma(\pi_1)$ of $\pi_1$ is depicted in Fig. 5b. It consists of four events $e_1 \triangleq T_1$ : x:=1, $e_2 \triangleq T_1$ : read z, $e_3 \triangleq T_2$ : y:=0, and $e_4 \triangleq T_2$ : x:=0. An arrow $e \to e'$ indicates that $\sigma(\pi_1)$ requires $e$ to occur before $e'$. Events of the same thread are ordered according to the program order of the respective thread. Events $e_1$ and $e_3$ are from different threads and write to the same variable, hence they are dependent and the section schedule needs to specify an ordering: $e_1$ must occur before $e_3$. Accordingly, the complete section schedule is $(\{e_1, e_2, e_3, e_4\}, \{(e_1, e_2), (e_3, e_4), (e_1, e_3)\})$.

By the following lemma, an execution from a state corresponding to the first node of a section and scheduled according to the respective section schedule will always lead to a state corresponding to the last node of the section. For instance, the following execution fragments both lead from the initial state to a state represented by $v_4$ $(s_4, s_4' \vDash \phi(v_4))$, as $e_1$ and $e_3$ are independent and can be swapped:

$$s_{init}, T_1, s_1, T_2, s_2, T_1, s_3, T_2, s_4 \leftrightsquigarrow e_1, e_3, e_2, e_4$$
$$s_{init}, T_2, s_1', T_1, s_2', T_1, s_3', T_2, s_4' \leftrightsquigarrow e_3, e_1, e_2, e_4$$

**Lemma 5 (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 $\pi$.*

A *program schedule* $\Sigma$ comprises several section schedules. $\Sigma$ is a labeled graph $(V_\Sigma, \to_\Sigma)$. 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. 5c 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. For example, the event $T_1 : y := 1$ from section $\pi_2$ must not occur in between events $T_1$ : read $z$ and $T_2 : y := 0$ from section $\pi_1$.

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 6 (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 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 5, 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 $\pi_0 \dots \pi_n$.*

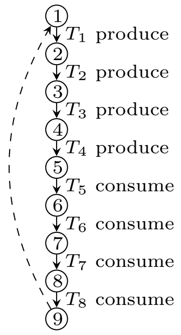


Fig. 6: First IVR for the producer-consumer problem (simplified)

```
Thread T1:                           Thread T2:
  while true:                          while true:
    lock(mutex1)                         lock(mutex2)
    lock(mutex2)                         lock(mutex1)
    execute_critical_section()           execute_critical_section()
    unlock(mutex2)                       unlock(mutex2)
    unlock(mutex1)                       unlock(mutex1)
```

Fig. 7: A program with a deadlock

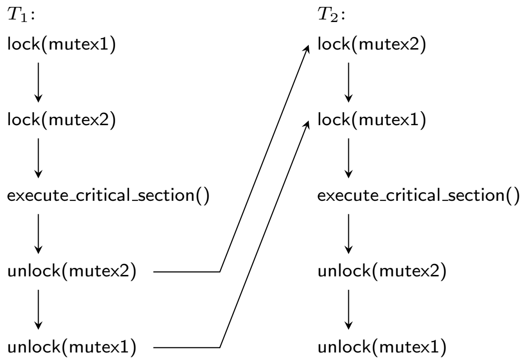


Fig. 8: Section schedule for the program of Fig. 7

## 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.[1] We refer to the (non-iterative) Impara

[1] 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

```
Threads
  T1: while true: produce()
  T2: while true: produce()
  T3: while true: consume()
  T4: while true: consume()

produce:
  if buffer_is_not_full():
    lock()
    assert buffer_is_not_full()
    add_item()
    unlock()

consume:
  if buffer_is_not_empty():
    lock()
    assert buffer_is_not_empty()
    remove_item()
    unlock()
```

Fig. 9 (a) The producer-consumer problem with a race condition

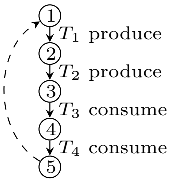


Fig. 9 (b) First IVR (simplified)

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

### 5.1 Implementation

To evaluate the enforcement of program schedules for infinite executions, we implement a custom (user space) scheduler.

In a first step, we automatically translate ARTs constructed by Impara-IMC to program schedules encoded as vector clocks. To omit sections in the generated program schedule that would never be executed and thereby reduce the size of the program schedule, we discard all paths in the ART that lead only to nodes labeled with *false*. As we use only deadlock-free ARTs, an alternative, feasible path, always exists. A given ART is traversed from the root. Recursively, we build section paths by traversing the graph until a branching node is reached. At the branching node, a fully expanded thread $T$ is chosen. The next sections are started at all child nodes of the branching node that are reached by a transition of $T$. For each section, the section schedule is generated based on the dependency information of memory accesses. Section schedules are represented by vector clocks. Additionally, each section schedule contains a link to all possible successor sections, i.e., those sections that start at a direct successor node of the current section. If there exist nodes $v$, $w$ such that all possible (interleaved) paths between $v$ and $w$ are equivalent and section paths, a single section path between $v$ and $w$ with relaxed scheduling constraints is sufficient. In this case, no dependencies between memory events need to be enforced. However, we use only the first IVR in our experiments (produced in a single iteration of Algorithm 1), hence we do not evaluate this case.

Firstly, all section schedules for the given ART are generated by enumerating them, including link information about successor sections, and marking the initial section.

Secondly, we instrument the source code of benchmark programs manually with callbacks to our user space scheduler and code for time measurement. The user space scheduler is implemented in C++11 and uses the C++ standard library for atomic memory operations. Program schedules are included as header files. Every access to a non-thread-local, global variable (shared variable) is replaced by a C++ preprocessor macro that calls the user space scheduler, executes the original statement, and calls the user space scheduler to notify that the statement has been executed. In our selection of benchmark programs, we had to instrument assignments and if-then-else statements. In the case of control flow branchings that depend on a shared variable, i.e., an if-then-else statement where the branching expression depends on a shared variable, additional callbacks are necessary to notify the scheduler of the taken control flow path.

To ensure that memory accesses enclosed by callbacks are indeed executed after the preceding callback and before the succeeding callback, memory fences are used.

The result of steps one and two is a multi-threaded program that executes concurrent memory accesses according to a given program schedule. Threads are executed concurrently and only forced to execute sequentially where required by the program schedule. Each time a thread $T$ enters the callback preceding a memory access, $T$ looks up the current section schedule and program counters of the other threads. If the vector clock of the section schedule, at the position of the current event of $T$, shows an event of an other thread that has to occur first, $T$ waits until this event has been executed. If no more events are required to occur before the current event of $T$ by the section schedule, $T$ executes the current memory access and, in the succeeding callback, updates its program counter so that the other threads are notified that $T$ has executed another event.

In case all events of the current section have already been executed, $T$ chooses the successor section associated to its current event. Waiting for all threads to completely execute the current section before switching to a successor section ensures that the program, at the end of each section, reaches a state that is represented by a node in the program schedule (and thereby, in the ART

programs so that Impara handles them correctly and their semantics is not changed. We will publish our modifications to Impara, including two bug fixes.

generated by the model checker). In case $T$ has no successor section associated to its current event, $T$ waits for an other thread to choose the next section. In case the last node of the current section is a branching node, only the thread with a control flow branching chooses the next section. In case $T$ has a control flow branching at the end of the last section, $T$ chooses the successor section based on the taken control flow branch.

Thirdly, we instrument the benchmark programs with code for time measurement. Each thread executes in an indefinite loop. Each time a thread has accomplished useful work in the current loop iteration, e.g., producing or consuming an item, writing a block or inode, or executing the critical section, it increments its *performance counter*. The main thread sleeps for 2 seconds, the time out duration, and subsequently prints the sum of the performance counters of all threads and terminates the program. Such a single run of a benchmark program is executed five times and we report the respective median value of performance counter sums. All experiments have been executed on a 4-core Intel Core i5-6500 CPU at 3.2 GHz.

While we manually instrumented the benchmark source code, an automated instrumentation is well conceivable. Main tasks of such an automated instrumentation are to identify shared variables and all points in the program, where dependent expressions are accessed. Relevant shared variables can be either overapproximated so that all shared or global variables are included or found by a static dependency analysis. Even if the variables to be instrumented are overapproximated, the expected additional execution time overhead is small, as our experiments show: a callback to our scheduler is fast if the current thread does not have to wait for other threads before executing the next variable access. Expressions that depend on a shared variable can likewise be found by a static dependency analysis. The automated instrumentation may of course be implemented on the level on the intermediate representation of a compiler and does not have to be conducted on the source code level.

### 5.2 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 in Fig. 6; it covers all executions in which the threads appear to execute their loop bodies atomically in the order $T_1, T_2, \ldots, T_8$. While the main bottleneck for Impara-C is state explosion and finding many coverings for different schedules, we observe that the main issue to produce $\mathcal{R}_1$ is to find a single covering that comprises all threads, i.e., to find a fair cycle. The essential predicates that lead to a fair cycle are:

$\mathsf{count} > 0$, $\mathsf{count} + 1 > 0$, $\mathsf{count} + 2 > 0$, $\mathsf{count} + 3 > 0$,
$\mathsf{count} \neq 1000$, $\mathsf{count} \neq 999$, $\mathsf{count} \neq 998$, $\mathsf{count} \neq 997$

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.

### 5.3 Deadlocks

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 in Fig. 7, 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. The corresponding program schedule consists of a single section schedule depicted in Fig. 8. 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.

### 5.4 Race conditions through erroneous synchronization

The program in Fig. 9a shows a variant of the producer-consumer problem with two producers and two con-

```
Variables:
  int block
  boolean busy
  boolean inode
  mutex m_inode
  mutex m_busy
Initially: inode = busy
```

```
Thread $T_1$:
  while true:
    lock(m_inode)
    if not inode:
      lock(m_busy)
      busy := true
      unlock(m_busy)
      inode := true
    block := 1
    unlock(m_inode)
```

```
Thread $T_2$:
  while true:
    lock(m_busy)
    if not busy:
      block := 0
    unlock (m_busy)
```

```
Thread $T_3$:
  while true:
    lock(m_inode)
    lock(m_busy)
    inode := false
    busy := false
    unlock(m_inode)
    unlock(m_busy)
```

Fig. 10: The file system benchmark

```
Thread $T_1$:
  while true:
    if not inode:
      busy := true
      inode := true
    atomic−begin
    assume inode and busy
    block := 1
    atomic−end
```

```
Thread $T_2$:
  while true:
    if not busy:
      atomic−begin
      assume not busy
      block := 0
      atomic−end
```

```
Thread $T_3$:
  while true:
    atomic−begin
    assume inode = busy
    inode := false
    busy := false
    atomic−end
```

Fig. 11: The file system benchmark with synchronization constraints in assume statements

```
Thread $T_2'$:
  while true:
    atomic−begin
    assume not busy
    block := 0
    atomic−end
```

Fig. 12: Thread $T_2'$: the if-statement is omitted

```
initially:
  empty buffer of size 1000
  count = 0
  mutex = 0

thread $T_{1...4}$:
  while true:
    lock()
    if count != 1000:
      int return_value = produce()
      assert(return_value != OVERFLOW);
    unlock()

thread $T_{5...8}$:
  while true:
    lock()
    if top > 0:
      return_value = consume();
      assert(return_value != UNDERFLOW);
    unlock()
```

Fig. 13: A correct program for the producer-consumer problem with four producers and four consumers

sumers which uses erroneous synchronization: both the produce and consume procedures 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 in Fig. 9b. 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.

### 5.5 Declarative synchronization

Fig. 10 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.

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 in Fig. 11. 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.

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 in Fig. 12. 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.

### 5.6 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).[2] 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.[3] Both Opt1 and Opt2 enable the concurrent execution of more memory accesses, e.g., because the beginning of a critical section can 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.

At the example of a section schedule of the producer-consumer benchmark with two threads, Fig. 14a–14b illustrates the difference between optimizations. Fig. 14a shows a section schedule for Opt0. All shared memory events are executed strictly sequentially, as it is the case with unconstrained executions: only the thread holding the lock is allowed to access shared memory. Opt1 removes the lock operations while maintaining the same ordering of events. Opt2, cf. Fig.14b, relaxes the original ordering, subsumes eight loop executions of both threads, and eliminates the redundant read event of count.

In Fig. 14b, when the consumer executes the scheduler callback before its first event (read count), it looks up the constraint $e_{12} \rightarrow e_{21}$ and waits for the producer to finish event $e_{12}$. When the producer in the callback after $e_{12}$ has notified that $e_{12}$ has been executed, the consumer continues and executes $e_{21}$. Similarly, the producer is permitted to execute $e_{14}$ before $e_{23}$ has been executed. Thus, the constrained execution under the optimized schedule permits "more" concurrency (i.e., more events to be executed concurrently) than the unconstrained execution with locks.

For instance, the consumer is allowed to read the counter already after the producer has written it and does not have to wait for the producer to also write an item to the buffer.

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. 5.3), 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. 5.5. 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. Subsequent IVRs were generated considerably faster than the first IVR, which might be caused by caching of facts in the model checker.

The verification time for the producer-consumer benchmark of both Impara-C and Impara-IMC appears to grow exponentially with the number of threads. This growth is not a limitation of our approach but a property of the application of lazy abstraction with interpolants in Impara. Potentially, Impara can be improved by including symmetry reduction, which would reduce the verification time for both Impara-C and Impara-IMC but is outside of the scope of this work.

Somewhat surprisingly, some benchmarks are slower when executed unconstrained than under Opt2. We conjecture that this is caused by more memory accesses being executed in parallel under Opt2, as all other effects of Opt2 only improve handling by our user space scheduler and do not affect unconstrained

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

[3] 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.

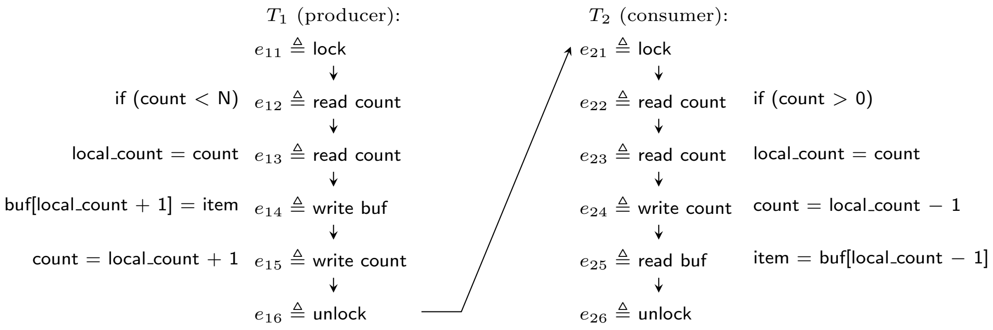

$T_1$ (producer):
$T_2$ (consumer):
$e_{11} \triangleq$ lock
if (count < N) $e_{12} \triangleq$ read count
local_count = count $e_{13} \triangleq$ read count
buf[local_count + 1] = item $e_{14} \triangleq$ write buf
count = local_count + 1 $e_{15} \triangleq$ write count
$e_{16} \triangleq$ unlock
$e_{21} \triangleq$ lock
$e_{22} \triangleq$ read count if (count > 0)
$e_{23} \triangleq$ read count local_count = count
$e_{24} \triangleq$ write count count = local_count − 1
$e_{25} \triangleq$ read buf item = buf[local_count − 1]
$e_{26} \triangleq$ unlock

Fig. 14 (a) Section schedule for the producer-consumer benchmark (Opt0)

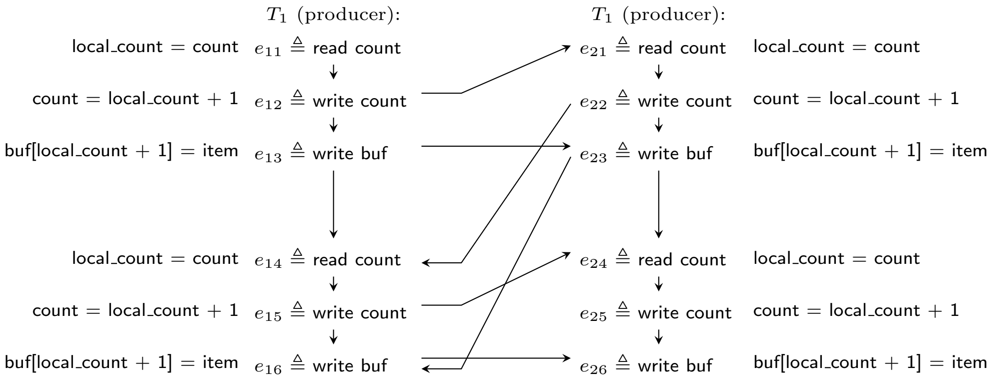

$T_1$ (producer):
$T_1$ (producer):
local_count = count $e_{11} \triangleq$ read count
count = local_count + 1 $e_{12} \triangleq$ write count
buf[local_count + 1] = item $e_{13} \triangleq$ write buf
local_count = count $e_{14} \triangleq$ read count
count = local_count + 1 $e_{15} \triangleq$ write count
buf[local_count + 1] = item $e_{16} \triangleq$ write buf
$e_{21} \triangleq$ read count local_count = count
$e_{22} \triangleq$ write count count = local_count + 1
$e_{23} \triangleq$ write buf buf[local_count + 1] = item
$e_{24} \triangleq$ read count local_count = count
$e_{25} \triangleq$ write count count = local_count + 1
$e_{26} \triangleq$ write buf buf[local_count + 1] = item

Fig. 14 (b) Section schedule for the producer-consumer benchmark (Opt2)

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 | 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 |

executions. It is, however, not directly possible to measure the effect of parallelizing memory accesses: in order to re-sequentialize memory accesses under Opt2, synchronization (e.g., over a mutex) would have to be added, which produces additional overhead.

In all cases but one, 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 unconstrained execution showed a performance of 13 567 135, which is still slower than Opt2.

Comparing the results for the producer-consumer benchmark with a buffer size of 1000 to those for a buffer size of 5, we observe that there is no considerable effect on Opt0–Opt2 but on most of the unconstrained executions. This observation is comprehensible, as the first IVR does not make use of more than at most four

| | 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 2: Experimental performance results for pfscan

cells in the buffer (in case of four producers). The performance of unconstrained executions decreases with a smaller buffer as the chance that the buffer is full and a producer has to wait is higher. For all three configurations with a buffer size of 5, Opt2 shows the highest execution time performance.

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[4], which show an overhead between 3% and 10% (with Opt2). Hence, IVRs can perform much better than input-covering schedules (60% overhead reported in [7]).

Tab. 2 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.

## 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) [21,28,40] 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 [13,24,2,3,26] 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, 27,31,35] 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 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

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

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 [37,25,14,22,33,34] 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 schedules. We state why it is legitimate to constrain scheduling (in contrast to inputs) and formulate general requirements on model checkers 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, and 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. 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)

23. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with wolverine. In: CAV, *LNCS*, vol. 6806, pp. 573–578. Springer (2011)
24. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: FMCAD. IEEE (2010)
25. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design **35**(1), 73–97 (2009)
26. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS, LNCS. Springer (2013)
27. Liu, T., Curtsinger, C., Berger, E.D.: Dthreads: efficient deterministic multithreading. In: SOSP. ACM (2011)
28. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV, LNCS. Springer (2006)
29. Metzler, P., Saissi, H., Bokor, P., Suri, N.: Quick verification of concurrent programs by iteratively relaxed scheduling. In: ASE. IEEE Computer Society (2017)
30. Metzler, P., Suri, N., Weissenbacher, G.: Extracting safe thread schedules from incomplete model checking results. In: SPIN, LNCS. Springer (2019)
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 multithreaded software with impact. In: FMCAD. IEEE (2013)