Int J Softw Tools Technol Transfer (2013) 15:413–431
DOI 10.1007/s10009-012-0232-3

Abstraction-guided synthesis of synchronization

Martin Vechev · Eran Yahav · Greta Yorsh

Published online: 23 May 2012

Abstract We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually. Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible. Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction. We implemented a prototype of our approach using numerical abstractions and applied it to verify several example programs.

Keywords Concurrency · Verification · Synthesis · Abstract interpretation

E. Yahav is a Deloro Fellow.
A preliminary version of this work appeared in the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

M. Vechev
ETH Zürich, Zurich, Switzerland

E. Yahav (✉)
Technion-Israel Institute of Technology, Haifa, Israel
e-mail: yahave@cs.technion.ac.il

G. Yorsh
ARM, Cambridge, UK

1 Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [8], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies S.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [2,7]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions—refining the abstraction and restricting program behaviors—to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Sect. 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g., [2,7]). These include moving through a pre-determined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [3]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. A solution of the atomicity constraints can be implemented by adding atomic sections to the program or by restricting the non-deterministic choices available to the scheduler.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Sect. 6, our approach provides a solution to a *quantitative synthesis* problem [4], as it can compute a *minimally atomic* safe schedule for a program, a schedule that poses minimal atomicity constraints on interleavings, and does not restrict interleavings unnecessarily.

Furthermore, our approach can be instantiated with different methods for: (i) modifying the program to eliminate invalid interleavings (ii) refining the abstraction (iii) choosing optimal solutions (quantitative criterion) (iv) implementing the resulting solution.

The problem we address in this paper is closely related to the ones addressed by program repair [11,13] and controller synthesis [23]. However, in contrast to these, our approach focuses on concurrent programs, uses abstract interpretation, and is able to handle infinite-state programs.

1.1 Main contributions

The contributions of this paper can be summarized as follows:

- We provide a novel algorithm for inferring correct and efficient synchronization in concurrent programs. The algorithm infers minimal atomic sections that can be verified under a given abstraction.
- We advocate a new approach to verification where both the program and the abstraction can be modified on the fly during the verification process. This enables verification of a restricted program where verification of the original program fails.
- We implemented our approach in a prototype tool called Guardian and applied it to synthesize synchronization for several interesting programs using numerical abstractions.

1.2 Limitations

Our focus in this paper is on the AGS algorithm (Sect. 4) and on an algorithm for eliminating invalid interleaving by adding atomic sections. In [16,17] we show how the same general idea can be applied to automatically infer memory fences.

While our approach can be instantiated with various abstraction-refinement algorithms and abstract domains, our current realization is only a first step:

- We use a simple abstraction-refinement approach.
- We integrate basic numerical abstract domains (to handle infinite-state numerical programs).
- We focus on safety specifications given as user-provided assertions.

Using more sophisticated refinement approaches, integrating additional abstract domains and handling liveness properties is left as future work.

2 Overview

In this section, we demonstrate our technique on a simple illustrative example. The discussion in this section is mostly informal, additional formal details are provided in Sect. 4. Additional examples are described in Sect. 7.

2.1 Example program

Consider the example shown in Fig. 1. In this example, the program executes three threads in parallel: `T1||T2||T3`. Different interleavings of the statements executed by these threads lead to different values being assigned to $y1$ and $y2$ (we assume that each individual statement executes atomically). In every execution of the program there is a single value assigned to $y1$ and a single value assigned to $y2$. The assertion in `T3` requires that the values of $y1$ and $y2$ are different. Initially, the values of all variables are 0.

For example, $y1$ gets the value 6, and $y2$ gets the value 2 in the interleaving `z++; x+=z; x+=z; y1=f(x); y2=x; z++; assert`. In the interleaving `x+=z; x+=z; y1=f(x); y2=x; z++; z++; assert`, $y1$ gets the value 5, and $y2$ gets the value 0.

Figure 2I shows the possible values of $y1$ and $y2$ that can arise during *all possible* program executions, assuming that the macro f executes atomically. Note that in some interleavings $y1$ and $y2$ may be evaluated for different values of x (i.e., x can be incremented between the assignment to $y1$ and the assignment to $y2$). The point $y1 = y2 = 3$ (marked in red in Fig. 2I) corresponds to values that violate the assertion. These values arise in the following interleaving: `z++; x+=z; y1=f(x); z++; x+=z; y2=x;assert`.

Our goal is to add efficient synchronization to the program such that no execution violates the assertion in `T3`.

The AGS algorithm iteratively eliminates invalid interleavings (under an abstraction) by either modifying the program or the abstraction. Figure 2 shows how the algorithm operates on the program of Fig. 1, and how it can move on both dimensions, choosing to modify either the program, or the abstraction, on every step. Before we explain Fig. 2, we explain how the algorithm modifies the program to eliminate invalid interleavings without any abstraction.

```
T1 {            T2 {        T3 {                 f(x) {
1: x += z       1: z++      1: y1 = f(x)          if (x==1)
2: x += z       2: z++      2: y2 = x               return 3;
}               }           3: assert             else if (x==2)
                                 (y1 ≠ y2)          return 6;
                            }                     else return 5;
                                                 }
```

Fig. 1 Simple example computing values of $y1$ and $y2$

2.2 Inferring synchronization

We begin by considering the example program without abstraction. Since this is an illustrative finite-state program, we can focus on the aspects of the algorithm related to generating atomicity constraints.

The algorithm accumulates atomicity constraints by iteratively eliminating invalid interleavings. Every invalid interleaving yields an atomicity constraint that describes *all*

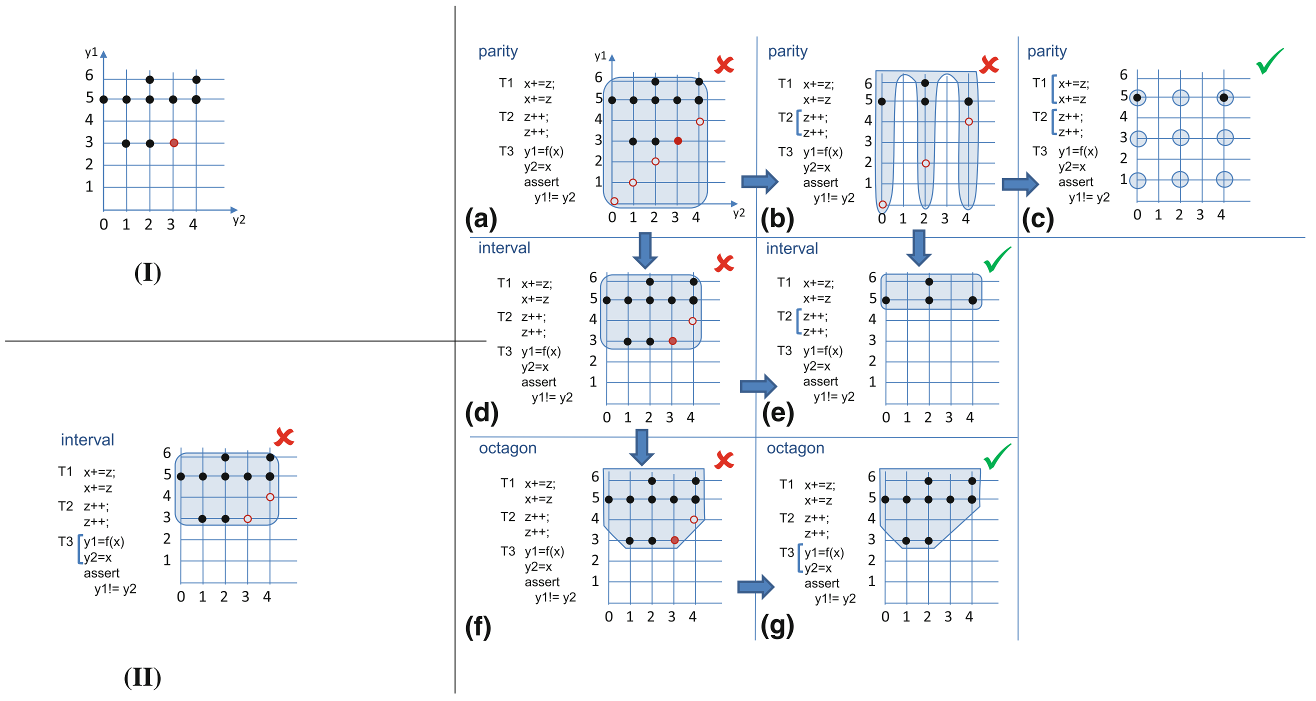


Fig. 2 **I** Values of $y1$ and $y2$ that arise in the program of Fig. 1. **II** Atomic section around the assignments to $y1$ and $y2$ under interval abstraction. **a–g** Possible steps of the AGS algorithm: on each step, the algorithm can choose between refining the abstraction (*down arrows*) and modifying the program by avoiding certain interleavings (*right arrows*)

possible ways to eliminate that interleaving, by disabling context-switches that appear in it.

The program of Fig. 1 has a single invalid interleaving `z++; x+=z; y1=f(x); z++; x+=z; y2=x; assert` corresponding to the point (3,3) in Fig. 2I. This interleaving can be eliminated by disabling either of the context switches that appear in it: the context switch between `x+=z` and `x+=z` in `T1`, between `z++` and `z++` in `T2`, and between `y1=f(x)` and `y2=x` in `T3`. This corresponds to the following atomicity constraint, generated by the AGS algorithm:

`[y1=f(x),y2=x]` $\vee$ `[x+=z,x+=z]` $\vee$ `[z++,z++]`

This constraint is a disjunction of three atomicity predicates, of the form `[s1,s2]`, where `s1` and `s2` are consecutive statements in the program. Each atomicity predicate represents a context-switch that can eliminate the invalid interleaving, and the disjunction represents the fact that we can choose either one of these three to eliminate the invalid interleaving. For this program, there are no additional constraints, and any satisfying assignment to this constraint yields a correct program. For example, one satisfying assignment is to set `[z++,z++]` to *true*. We can then implement this assignment by adding an atomic section around `z++` and `z++` in `T2`, yielding a correct program.

Since we can obtain multiple solutions, it is natural to define a quantitative criterion for choosing among them. This criterion can be based on the number of atomic sections, their length, etc. Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. In this example, each of the three possible solutions only requires a single atomic section of two statements.

Next, we illustrate how AGS operates under abstraction. In this example, we use several numerical domains: parity, intervals, and octagon. In Sect. 7, we show refinement by increasing the set of variables for which the abstraction tracks correlations.

2.3 Inferring synchronization under parity abstraction

We first show how the algorithm works using the parity abstraction over $y1$ and $y2$. The parity abstraction represents the value of a variable by its parity. Variables $y1$ and $y2$ take abstract values from $\{\bot, E, O, \top\}$ with the standard meaning.

The starting point, parity abstraction of the original program, is shown in Fig. 2a. It shows the concrete values of $y1$ and $y2$ that can arise during program execution, and their abstraction. The concrete values are shown as full circles and are the same as in Fig. 2I. Black circles denote the concrete values that satisfy the assertion, and red circle values that violate the assertion. The shaded area denotes the concretization of the abstract values computed for $y1$ and $y2$. The abstract values for both $y1$ and $y2$ are $\top$. As a result, the concretization (the shaded area) covers the entire plane. In particular, it covers concrete values that violate the assertion. Values that cannot arise in any concrete execution of the program (false alarms) are shown as hollow red circles in the figure.

The AGS algorithm performs abstract interpretation of the program from Fig. 1 using parity abstraction. In Fig. 3 we show a part of the abstract transition system constructed by AGS. Figure 3 only shows abstract states that can reach an error state. Error states are shown as dashed red line circles in the figure.

The values of variables in a state are shown as a tuple $\langle pc_1, pc_2, pc_3, x, z, y1, y2\rangle$, where variables $y1$ and $y2$ take an abstract value from the parity domain.

States that differ only in their values for $y1$ and $y2$ are merged into a single abstract state. When joining two abstract values O and E, the resulting value is $\top$, representing the fact

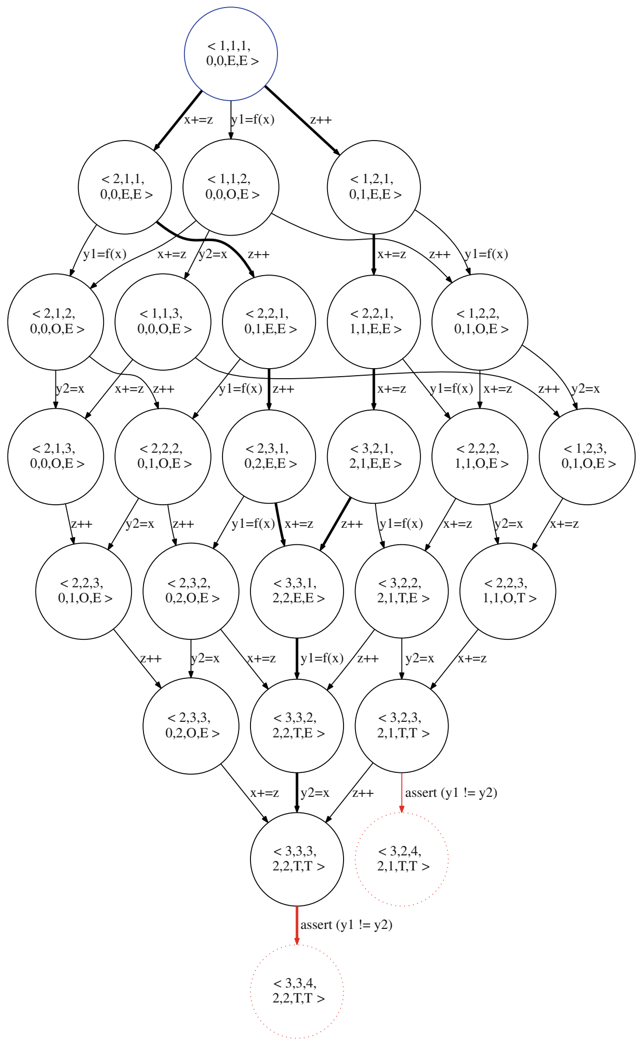


Fig. 3 Partial abstract transition system for the program of Fig. 1. Only abstract states that can reach an error state are shown

that the value may be either even or odd. For example, the transition `y1=f1(x)` from state $\langle 3, 2, 1, 2, 1, E, E\rangle$ updates the value of $y1$ to be even, which should lead to an abstract state $\sigma_1 = \langle 3, 2, 1, 2, 1, E, E\rangle$. The transition `x+=z` from state $\langle 2, 2, 2, 1, 1, O, E\rangle$ updates x and should lead to an abstract state $\sigma_2 = \langle 3, 2, 1, 2, 1, O, E\rangle$. Because σ_1 and σ_2 differ only in the abstract value for $y1$, they are joined into a single abstract state $\langle 3, 2, 2, 2, 1, \top, E\rangle$.

This abstract transition system is very simple and in particular contains no cycles; however, this is only for illustrative purposes. The AGS algorithm handles all forms of abstract transition systems.

Under parity abstraction, there are several invalid interleavings. The choice which of them to eliminate first is important, as discussed in Sect. 5. The AGS algorithm first chooses to eliminate the invalid interleaving: $\pi_1 =$ `z++; x+=z; x+=z; z++; y1=f(x); y2=x; assert`. This interleaving is shown in Fig. 3 by emphasizing its edges (the right emphasized path). Under this interleaving, and under the parity abstraction, the values of $y1$ and $y2$ at the point of the assertion $\langle 3, 3, 3, 2, 2, \top, \top\rangle$ are both $\top$, meaning that the assertion *may* be violated.

The AGS algorithm can now choose whether to try and eliminate the invalid interleaving by either adding atomicity, or by refining the abstraction. Figure 2 shows these alternatives, which we explain in detail in the rest of this section.

Eliminate π_1 by atomicity constraint To eliminate this interleaving, the following constraint is generated: `[z++,z++]`. This step is shown as the step from Fig. 2a–b. Note that the program in Fig. 2b has an atomic section around the statements `z++` and `z++` in `T2`. This limits the concrete values that $y1$ and $y2$ can take, as shown by the full circles in Fig. 2b, compared to those on Fig. 2a. In particular, it eliminates the error state in which $y1$ and $y2$ both have the value 3 (no red full circle in the figure).

However, parity abstraction is not yet precise enough to verify the correctness of the resulting program, as shown by the shaded area in Fig. 2b. During abstract interpretation of the program, $y1$ takes both the values E and O, and thus goes to $\top$. The concretization (the shaded area) therefore spans all possible concrete values of $y1$. The abstract value of $y2$ remains E, therefore the concretization (the shaded area) only contains even values of $y2$. The abstract values represent three points that violate the assertion, shown as hollow red circles in Fig. 2b.

After eliminating π_1 by adding the constraint `[z++,z++]`, the following (abstract) interleaving may violate the assertion: $\pi_2 =$ `x+=z;z++;z++;x+=z;y1=f(x);y2=x;assert`. This interleaving yields the abstract values $y1 = \top$ and $y2 = \top$ at the point of the assertion, meaning that the assertion may be violated. The interleaving π_2 is shown in Fig. 3 as the left emphasized path.

Eliminate π_2 by atomicity constraint To eliminate this interleaving, the following constraint is generated: `[x+=z,x+=z]`. This step is shown as the step from Fig. 2b–c. The resulting overall constraint is: `[x+=z,x+=z]` $\wedge$ `[z++,z++]`

With this atomicity constraint, under the parity abstraction, there are no further invalid interleavings. This constraint is satisfied by a program that has the statements `x+=z` and `x+=z` of `T1` execute atomically, and the statements `z++` and `z++` of `T2` execute atomically. In this program, the abstract values are $y1 = O$ and $y2 = E$. These abstract values guarantee that the assertion is not violated, as shown in Fig. 2c.

Eliminate π_2 by changing the abstraction After eliminating the interleaving π_1, all remaining concrete interleavings satisfy the assertion, but we could not prove it under parity abstraction. Instead of eliminating interleaving π_2 by adding atomicity constraints, as described above, we can choose to change the abstraction from parity to interval, moving from Fig. 2b–e. Interval abstraction is precise enough to prove this program.

2.4 Inferring synchronization under interval abstraction

Instead of eliminating interleaving π_1 by adding an atomicity constraint, the algorithm can choose to try and eliminate π_1 by refining the abstraction from parity to interval. This corresponds to the step from Fig. 2a–d. Under interval abstraction, the abstract values are $y1 = [3, 6]$ and $y2 = [0, 4]$, representing two points that may violate the assertion, as shown in figure Fig. 2d.

The algorithm can again choose to eliminate invalid interleavings by adding an atomicity constraint (step from Fig. 2d–e) or by abstraction refinement (step from Fig. 2d–f). In the former case, AGS produces the overall constraint:

(`[x+=z,x+=z]` $\vee$ `[z++,z++]`)
 $\wedge$ (`[y1=f(x),y2=x]` $\vee$ `[x+=z,x+=z]` $\vee$ `[z++,z++]`)

This constraint requires only one of `T1` and `T2` to execute atomically. Figure 2e shows a program corresponding to one of the solutions, in which `T2` is atomic.

As apparent from the constraint above, `[y1=f(x),y2=x]` is not sufficient for showing the correctness of the program under the interval abstraction. The result of applying interval abstraction to the program implemented from this constraint is shown in Fig. 2II.

2.5 Inferring synchronization under the octagon abstraction

Finally, the octagon abstract domain [21], maintains enough information to only require atomicity as in the case with full information. In particular, it is sufficient to make `y1 = f(x)` and `y2 = x` execute atomically for the program to be successfully verified under the Octagon abstraction, as shown in Fig. 2g.

3 Preliminaries

A transition system ts is a tuple $\langle \Sigma, T, Init\rangle$ where Σ is a set of states, $T \subseteq \Sigma \times \Sigma$ is a set of transitions between states, and $Init \subseteq \Sigma$ are the initial states. For a transition $t \in T$, we use $src(t)$ to denote the source state of t, and $dst(t)$ to denote its destination state.

For a transition system ts, a trace π is a (possibly infinite) sequence of transitions $\pi_0, \pi_1, \ldots$ such that for every $i > 0$, $\pi_i \in T$ and $dst(\pi_{i-1}) = src(\pi_i)$. For a finite trace π, $|\pi|$ denotes its length (number of transitions). We use $t.\pi$ to denote the trace created by concatenation of a transition t and a trace π, when $dst(t) = src(\pi_0)$.

A complete trace π is a trace that starts from an initial state: $src(\pi_0) \in Init$. We use $[\![ts]\!]$ to denote the (prefix-closed) set of complete traces of transition system ts.

Program syntax We consider programs written in a simple programming language with assignment, non-deterministic choice, conditional goto, sequential composition, parallel composition, and atomic sections. The language forbids dynamic allocation of threads, nested atomic sections, and parallel composition inside an atomic section. Note that a program can be statically associated with the maximal number of threads it may create in any execution. Assignments and conditional goto statements are executed atomically. All statements have unique labels. For a program label l, we use $stmt(l)$ to denote the unique statement at label l.

We use Var to denote the set of (shared) program variables. To simplify the exposition, we do not include local variables in definitions, although we do use local variables in examples. There is nothing in our approach that prevents us from using local variables, but having local variables makes the formal definitions cumbersome. We assume that all program variables have integer values, initialized to 0.

Program semantics Let P be a program with variables Var. Let k be the maximal number of threads in P, with thread identifiers $1, \ldots, k$. A state s is a pair $\langle val_s, pc_s\rangle$ where $val_s\colon Var \to Int$ is a valuation of the variables, and $pc_s\colon \{1, \ldots, k\} \to Int$ is the program counter of each thread, which ranges over program labels in the code executed by the thread. We define a transition system for a program P to be $\langle \Sigma_P, T_P, Init_P\rangle$, where transitions T_P are labeled by program statements. For a transition $t \in T_P$, we use $stmt(t)$ to denote the corresponding statement. We use $lbl(t)$ and $tid(t)$ to denote (unique) program label and thread identifier that correspond to $stmt(t)$, respectively.

A transition t is in T_P if all of the following conditions hold:
(a) the program counter of the thread $tid(t)$ in state $src(t)$ is at program label $lbl(t)$,
(b) the execution of the statement $stmt(t)$ from state $src(t)$ by thread $tid(t)$ results in state $dst(t)$,
(c) no thread except the one executing transition t is inside an atomic section in state $src(t)$.

We use $[\![P]\!]$ to denote the set of traces of P, i.e., $[\![P]\!] = [\![ts]\!]$ where $ts = \langle \Sigma_P, T_P, Init_P\rangle$.

Abstraction Our method is based on abstract interpretation [8]. In this section, we quickly review relevant terminology that will be used throughout the paper.

An abstract domain is a complete join semilattice $A = \langle A, \sqsubseteq, \sqcup, \bot\rangle$, i.e., a set A equipped with a partial order $\sqsubseteq$, such that for every subset X of A, A contains a least upper bound (or join), denoted $\sqcup X$. The bottom element $\bot \in A$ is $\sqcup\, \emptyset$. We use $x \sqcup y$ as a shorthand for $\sqcup\{x, y\}$.

In this paper, we assume that the abstract domain A is a powerset of abstract states, with (partially) disjunctive join. An abstract state s is ranging over an abstract domain $B = \langle B, \sqsubseteq_B, \sqcup_B, \bot_B\rangle$. We use s for the singleton $\{s\}$ when no confusion is likely.

For $X \subseteq \Sigma_P$, the abstraction function α is defined by $\alpha(X) = \sqcup\{\beta(s) \mid s \in X\}$, where β is the abstraction function for the underlying domain of abstract states. For a given β, the abstraction α can vary anywhere on the range between "relational" and "cartesian", depending on the definition of $\sqcup$.

An abstract transformer for a program statement st is denoted by $[\![st]\!]_\alpha\colon A \to A$. For $a \in A$, the abstract transformer is defined pointwise: $[\![st]\!]_\alpha(a) = \sqcup\{[\![st]\!]_\beta(\sigma) \mid \sigma \in a\}$, where $[\![st]\!]_\beta$ is the abstract transformer for the underlying domain of abstract states.

We abuse the notation slightly and use α to collectively name all the components of an abstract interpreter: its abstract domain, including the underlying domain of abstract states, abstract transformers, and widening operator, if defined.

We define an abstract transition system for P and α to be $\langle \Sigma_P^\natural, T_P^\natural, Init_P^\natural\rangle$, where $\Sigma_P^\natural$ is a set of abstract states, $Init_P^\natural = \alpha(Init_P)$, and a transition (σ, σ') labeled by a program statement st is in $T_P^\natural$ if and only if $[\![st]\!]_\beta(\sigma) \sqsubseteq_B \sigma'$.

We use $[\![P]\!]_\alpha$ to denote the set of abstract traces of P, i.e., $[\![P]\!]_\alpha = [\![ts]\!]$ where ts is the abstract transition system for P and α, in which $\Sigma_P^\natural$ is the result of abstract interpretation, i.e., the set of abstract states at fixed point.

Specification The user can specify a state property S, which describes a set of program states. This property can refer to program variables and to the program counter of each thread (e.g., to model local assertions). Our approach can be extended to handle any temporal safety specifications, expressed as a property automaton, by computing the synchronous product of program's transition system and the property automaton [32].

Given a (concrete or abstract) state s, we use $s \models S$ to denote that the state s satisfies the specification S. We lift it to traces as follows. A trace π satisfies S, denoted by $\pi \models S$, if and only if $src(\pi_0) \models S$ and for all $i \geq 0$, $dst(\pi_i) \models S$. A set Π of (concrete or abstract) traces satisfies S, denoted by $\Pi \models S$ if and only if all traces in it satisfy S.

4 Abstraction guided synthesis

In this section, we describe the abstraction-guided synthesis algorithm at a high-level. In Sect. 5, we provide a full realization of the algorithm with specific design choices.

Goal Given an input program, a specification, and an abstraction, the goal of the AGS algorithm is to produce a (possibly modified) program that satisfies the specification.

The idea As explained in Sect. 2, the AGS algorithm iteratively eliminates invalid interleavings (under an abstraction) by either modifying the program or the abstraction. When the algorithm terminates successfully (without aborting), it is guaranteed that all invalid interleavings of the program have been eliminated by additional synchronization. For example, given the program of Fig. 1, and parity abstraction, the AGS algorithm produces the program of Fig. 2c by adding two atomic sections to the original program. The process is described informally in Sect. 2.3.

The algorithm Algorithm 1 provides a declarative description of abstraction-guided synthesis. The main loop of the algorithm selects an abstract trace π of the program P such that π satisfies the atomicity formula φ, but does not satisfy the specification S. We use Γ to denote a satisfying assignment for φ, and $P \mid_\Gamma$ to denote the program P with Γ realized syntactically in P (see Sect. 4.2). After selecting an invalid abstract trace π, the algorithm attempts to eliminate this invalid interleaving π by either:

- adding atomicity constraints: the procedure `avoid` generates atomicity constraints that disable π. The constraints generated by `avoid` for π are accumulated by AGS in the formula φ.
- refining the abstraction: using a standard abstraction refinement approach (e.g., [2,7]) to refine the abstraction.

On every iteration, the algorithm takes into account the updated φ and α when choosing an invalid interleaving π.

Some of the (abstract) invalid interleavings may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. The choice of whether to eliminate an interleaving via abstraction refinement, or by adding atomic sections, is discussed in Sect. 4.5.

When all invalid interleavings have been eliminated, AGS calls the procedure `implement` to find a solution for the constraints accumulated in φ.

Algorithm 1: Abstraction-Guided Synthesis.

```
   Input: Program $P$, Specification $S$, Abstraction $\alpha$
   Output: Program satisfying $S$ under $\alpha$ or abort
1  $\varphi = true$
2  while $true$ do
3      $\Pi = \{\pi \mid \pi \in [\![P \mid_\Gamma]\!]_\alpha, \Gamma \models \varphi, \pi \not\models S\}$
4      if $\Pi$ is empty then return implement($P$,$\varphi$)
5      $\pi$ = select trace from $\Pi$
6      if $shouldAvoid(\pi, \alpha)$ then
7          $\psi$ = avoid($\pi$)
8          if $\psi \neq false$ then $\varphi = \varphi \wedge \psi$
9          else abort
10     else
11         $\alpha'$ = refine($\alpha, \pi$)
12         if $\alpha' \neq \alpha$ then $\alpha = \alpha'$
13         else abort
14     end
15 end
```

Function `avoid(`π`)`

```
   Input: Trace $\pi$
   Output: Atomicity constraint for avoiding $\pi$
1  $\rho = false$
2  foreach $i = 0, \ldots, |\pi|$ do
3      if exists $j > i + 1$ such that $tid(\pi_i) = tid(\pi_j)$ and
4      for all $l$ such that $i < l < j$, $tid(\pi_i) \neq tid(\pi_l)$
5      then $\rho = \rho \vee [lbl(\pi_i), lbl(\pi_j)]$
6  end
7  return $\rho$
```

Function `implement(`P, φ`)`

```
   Input: Program $P$, atomicity formula $\varphi$
   Output: Program with atomic sections satisfying $\varphi$
   Find a minimal satisfying assignment $\Gamma \models \varphi$
   return $P \mid_\Gamma$
```

Example 1 Figure 3 shows a part of the abstract transition system constructed by AGS for the program of Fig. 1 under parity abstraction. The values of variables in a state are shown as a tuple $\langle pc_1, pc_2, pc_3, x, z, y1, y2\rangle$, where variables $y1$ and $y2$ take an abstract value from the parity domain.

In the first iteration, the algorithm picks the invalid interleaving: π_1 = `z++; x+=z; x+=z; z++; y1=f(x); y2=x; assert`. The algorithm computes $avoid(\pi_1)$ = `[z++,z++]`. The constraint $avoid(\pi_1)$ is added to the global constraint φ, and the algorithm moves to the next iteration. After avoiding π_1, there is still an invalid interleaving π_2 = `x+=z;z++;z++;x+=z;y1=f(x);y2=x;assert`. Note that π_2 satisfies the constraint `[z++,z++]`, as there is no context switch between these statements in π_2. The computes

$avoid(\pi_2)$ = `[x+=z,x+=z]` The constraint $avoid(\pi_2)$ is added to the global constraint φ, resulting in: `[x+=z,x+=z]` $\wedge$ `[z++,z++]` With this atomicity constraint, under the parity abstraction, there are no further invalid interleavings, and the algorithm therefore returns $implement(P, \varphi)$. This yields the program of Fig. 2c.

Algorithm 1 is parametric on three dimensions:

- The language of constraints can be changed by modifying procedure `avoid`(π), Algorithm 1.
- The implementation of constraints in terms of the (concurrency) statements available in the specific programming language can be changed by modifying procedure `implement`(P,φ).
- Abstraction refinement approaches can be plugged in by adjusting procedure `refine`(α, π).

For generality, the algorithm is described here in terms of the trace semantics, but it is easy to construct a variant that operates over states (e.g., see [15–17]). The formulation over the trace semantics enables the use of the same algorithm in a dynamic setting where traces are obtained by executing the program (e.g., [18]). Constraints obtained from (concrete) dynamic executions can be used to infer synchronization that avoids future violations of the specification. Such constraints can be then taken into account by a benevolent runtime ("online") or by changing the program ("offline").

4.1 Generating atomicity constraints

The procedure `avoid` (used in Line 1 of Algorithm 1) takes a trace π as input, and generates an atomicity constraint that describes all context switches in π, and thus describes all possible ways to eliminate π by adding atomic sections to the original program.

The atomicity constraint generated by `avoid` is a disjunction of atomicity predicates. An atomicity predicate requires that a pair of consecutive program statements execute atomically, without interleaving execution of other threads between them.

Atomicity predicates and atomicity constraints Formally, given a program P, and a pair of program labels l and l', we use $[l, l']$ to denote an *atomicity predicate*. In our examples, we write $[stmt(l), stmt(l')]$ instead of $[l, l']$. An *atomicity formula* is a conjunction of disjunctions of atomicity predicates. Let π be a trace in a (concrete or abstract) transition system of P. We say that π satisfies $[l, l']$, denoted by $\pi \models [l, l']$, if and only if for all $0 \leq i$, if $lbl(t_i) = l$ and $i + 1 < |\pi|$, then $lbl(t_{i+1}) = l'$ and $tid(t_i) = tid(t_{i+1})$.

A set of traces Π satisfies an atomicity predicate p, denoted by $\Pi \models p$, if and only if all the traces in Π satisfy p. Similarly, we interpret conjunctions and disjunctions of atomicity predicates as intersection and union of sets of traces. The set of traces that satisfy an atomicity formula φ is denoted by $[\![\varphi]\!]$.

The function `avoid` The function `avoid` only generates atomicity predicates for neighboring locations (locations that appear in the same thread, where one location immediately follows the other), with the intuitive meaning that no operation is allowed to interleave between the execution of these neighboring locations.

The algorithm identifies all context switches in π as follows. A context switch after transition π_i occurs if there is another transition π_j by the same thread later in the trace, but not immediately after π_i (hence the condition $j > i + 1$ in Line 3). Then, if the transition π_j is the first such transition after π_i (as checked in Line 4), we generate the atomicity predicate $[lbl(\pi_i), lbl(\pi_j)]$ (Line 5). We repeat this process in a loop over all context switches in π.

Example 2 The interleaving π =`z++; x+=z; y1=f(x); z++; x+=z; y2=x; assert` of the program of Fig. 1 can be eliminated by disabling either of the context switches that appear in it: the context switch between `x+=z` and `x+=z` in `T1`, between `z++` and `z++` in `T2`, and between `y1=f(x)` and `y2=x` in `T3`. This corresponds to the following atomicity constraint, generated by $avoid(\pi)$:

`[y1=f(x),y2=x]` $\vee$ `[x+=z,x+=z]` $\vee$ `[z++,z++]`

Each atomicity predicate represents a context-switch that can eliminate π, and the disjunction represents the fact that we can choose either one of these three to eliminate π.

In the case of an invalid sequential interleaving, an interleaving in which each thread runs to completion before it context-switches to another thread, it is (obviously) impossible to avoid the interleaving by adding atomic sections. In such cases, `avoid` returns $false$ and AGS aborts.

4.2 Implementing atomicity constraints

The procedure `implement` takes a program P and an atomicity formula φ as input. An atomicity formula can be seen as a formula in propositional logic, where the atomicity predicates are treated as propositional (boolean) variables. Note that the atomicity formula is in positive CNF, and thus it is always satisfiable.

The procedure constructs a program P' by finding a minimal satisfying assignment for φ, i.e., a satisfying assignment with the smallest number of propositional variables set to *true*. The atomicity predicates assigned to true in that assignment are then implemented as atomic sections in the program.

Our approach separates the characterization of valid solutions from their implementation. The atomicity formula φ maintained in the algorithm provides a symbolic description of possible solutions. In this paper, we choose to realize

```
T1 {                 | T2 {
 while (*) {         |  if (x==1) {
  x++                |   assert false
  x++                |  }
 }                   | }
}                    |
```

Fig. 4 Limitations of implementability. Correctness only requires the first iteration of the loop in T1 be executed atomically. Implementability forces every iteration to be executed atomically

these by changing the program and adding atomic sections. However, these could be realized using other synchronization mechanisms, as well as by controlling the scheduler of the runtime environment (if such a scheduler exists).

In general, there could be multiple satisfying assignments for φ, corresponding to different additions of atomic sections to the input program P. Usually, we are interested in minimal satisfying assignments, as they represent solutions that do not impose redundant atomic sections.

To realize a satisfying assignment $\Gamma \models \varphi$ as atomic sections, we find minimal (contiguous) atomic sections that respect the assignment. The program obtained from P by adding these atomic sections is denoted by $P\ |_\Gamma$. To find minimal atomic sections for Γ, we construct the set of program labels in which context switches are not permitted by Γ: $L = \{l' \mid [l, l'] \in \Gamma\}$. For every maximally-connected component of L in the control-flow graph of the original program, we find the immediate dominator and postdominator, and add (begin and end) atomic section at these labels, respectively. This may cause extra statements included in an atomic section, eliminating additional interleavings. This situation is sometimes unavoidable when implementing atomicity constraints using atomic sections.

It is possible that implementing an assignment Γ results in eliminating additional interleavings even when there are no extra statements in the atomic section. Consider the example of Fig. 4. In this example, T2 cannot interleave with the first iteration of the loop in T1. But once the first iteration is over, it can interleave with any other iteration. However, since we require implementation via atomic sections, the only implementable solution is to add an atomic section around the statements x++ and x++ inside the loop, forcing every iteration of the loop to be executed atomically.

4.3 Abstraction refinement

The procedure refine takes an interleaving π as input and attempts to refine the abstraction in order to avoid π. For that to be possible, π has to be an artifact of the abstraction, and not correspond to a concrete invalid interleaving. AGS tries to refine the abstraction by calling refine, but if the abstraction cannot be refined, and refine returns the same abstraction, AGS aborts.

In this paper, we focus on the procedure for restricting invalid interleavings, and can leverage any standard refinement scheme (e.g., [2,3,7,25]). In the examples, we use two kinds of simple refinements: one that moves to another abstract domain (Sect. 2), and one that varies the set of variables that are abstracted relationally (Sect. 7).

4.4 Choosing interleaving π to eliminate

Since our program modifications consist of adding atomic sections, we cannot eliminate sequential executions (which have no context switches). It is therefore required that we can verify the correctness of the sequential runs of the program under the given abstraction.

In fact, for verifying the correctness of interleavings that involve fewer context-switches, less precise abstractions can be sufficient.

Generally, it is natural to consider interleavings in an increasing order of the number of context switches. Atomicity constraints obtained for interleavings with a lower number of context switches restrict the space that needs to be explored for interleavings with higher number of context switches.

4.5 Program modification vs. abstraction refinement

When an invalid interleaving π is detected, a choice has to be made between refining the abstraction and adding an atomicity constraint that eliminates π. This choice is denoted by the condition $shouldAvoid(\pi, \alpha)$ in the algorithm. Apart from clear boundary conditions outlined below, this choice depends on the particular abstractions with which the algorithm is used.

When π is a sequential interleaving, and avoid is realized as the addition of atomic sections, it is impossible to add atomicity constraints to avoid π. Therefore, in this case, the only choice is to refine the abstraction (if possible). Hence, the condition $shouldAvoid(\pi, \alpha)$ is set to return $false$ when π is a sequential interleaving.

Similarly, depending on the refinement framework used, it may be impossible to further refine the abstraction α. For example, when using a fixed sequence of abstraction with increasing precision (as in Sect. 2), upon reaching the most precise abstraction in the sequence, it is not possible to further refine the abstraction. Therefore, in this case, the only choice is trying to avoid the interleaving π, and the condition $shouldAvoid(\pi, \alpha)$ returns $true$ when it is known a priori that α cannot be refined anymore.

For refinement schemes that use symbolic backwards execution to find a concrete counterexample (e.g., [2,7]), the condition $shouldAvoid(\pi, \alpha)$ can be based on the result of the symbolic execution. When the refinement scheme is able to find a concrete counterexample, $shouldAvoid(\pi, \alpha)$ can choose to repair, using the concrete counterexample as basis.

If the refinement scheme fails to find a concrete counterexample, but also fails to find a spurious path for abstraction refinement, $shouldAvoid(\pi, \alpha)$ can again choose to repair, as refinement cannot be applied.

Attempting verification with a refined abstraction may fail due to state explosion. In most cases it is impossible to check for such failure a priori in the condition $shouldAvoid(\pi, \alpha)$. Practically, it is useful to invoke the verification procedure as a separate task, and implement a backtracking mechanism for the refinement when verification fails to terminate after a certain time. Backtracking the refinement may enable successful verification of a more constrained variant of the program.

5 A realization of abstraction guided synthesis

In the previous section, we described the AGS algorithm in a declarative manner, and omitted some details that we now address: (i) how do we compute $[\![P \;|_{\Gamma}]\!]_{\alpha}$ where $\Gamma \models \varphi$? (ii) how do we obtain an interleaving $\pi \not\models S$?

Algorithm 2: Realization of Abstraction-Guided Synthesis.

```
   Input: Program $P$, Specification $S$, Abstraction $\alpha$
   Output: Program satisfying $S$ under $\alpha$ or abort
1  $states = workset = Init^{\natural}_P$
2  $\varphi = true$
3  while $workset$ is not empty do
4      $\sigma =$ select and remove state from $workset$
5      foreach $Statement\ st$ do
6          if $enabled(st, \sigma, \varphi, states)$ then
7              $\sigma' = [\![st]\!]_{\beta}(\sigma)$
8              if $\sigma' \not\models S$ then
9                  select
                   $\pi \in \mathtt{Traces}(Init^{\natural}_P, \{\sigma'\}, states \setminus workset, \varphi)$
10                 if $shouldAvoid(\pi, \alpha)$ then
11                     $\psi = avoid(\pi)$
12                     if $\psi \neq false$ then
13                         $\varphi = \varphi \wedge \psi$
14                         $states = workset = Init^{\natural}_P$
15                         $disabled = \emptyset$
16                     else abort
17                 else
18                     // $refine(\pi)$
19                 end
20             else
21                 if $\{\sigma'\} \not\sqsubseteq states$ then
22                     $states = states \sqcup \{\sigma'\}$
23                     $X = \{\sigma'' \in states \mid \sigma' \sqsubseteq_B \sigma''\}$
24                     $workset = workset \sqcup X$
25                 end
26             end
27         end
28     end
29 end
30 return $implement(P,\varphi)$
```

5.1 An optimized algorithm for abstraction-guided synthesis

Algorithm 2 is an optimized version of Algorithm 1. In the optimized algorithm, we focus on the exploration code, and on the code for avoiding an interleaving (Lines 11–16), the code for refinement is similar and is abbreviated to a comment in Line 18.

The algorithm combines (forward) abstract interpretation of the program, with (backward) exploration of invalid interleavings. The abstract interpretation part of the algorithm is standard, and uses a workset to maintain abstract states that should be explored. Once the workset is empty we know a fixed point is reached.

The main idea of the algorithm is to use the constraints accumulated in φ to restrict the space that has to be explored both forward and backward. In particular, this optimization avoids constructing the entire (unrestricted) transition system upfront.

Rather than naively enumerating *all* acyclic traces in the abstract transition system, we first use standard abstract interpretation to compute the set $states$ of abstract states reachable from $Init^{\natural}_P$ under abstraction α. Using the reachable states, we explore invalid interleavings and eliminate them, while taking φ into account. The algorithm is amenable to several optimizations, and we describe them later in this section.

We begin by describing the details of the forward exploration, which explores reachable states. The details of backward exploration—enumerating traces—are described in Sect. 5.2.

Forward pruning when exploring states Forward exploration of new states is restricted by the constraints accumulated in φ (Line 6). For every invalid interleaving π, the formula φ represents all the possible ways to eliminate π. This means that the algorithm only restricts further exploration when the next exploration step contradicts *all possible ways* to eliminate existing invalid interleavings.

Technically, this is done by defining the notion of an *enabled statement* with respect to a given state and atomicity constraint. We say that statement st is *φ-enabled in state σ* when executing st from σ does not contradict φ. Formally, given a set of states V, the condition $\mathtt{enabled}(st, \sigma, \varphi, V)$ holds if and only if for every pair of transitions t and t' such that $src(t) \in V$, $dst(t') \in V$, $dst(t) = src(t') = \sigma \in V$ and $stmt(t') = st$, the partial trace $t.t'$ satisfies φ.

For example, if φ is `[a,c]` then st is not φ-enabled in σ, in the partial state space shown in Fig. 5. However, if φ is `[a,c]` $\vee$ `[b,c]`, then st is φ-enabled in σ.

Adding explored states The set $states$ of reachable abstract states in the algorithm is not guaranteed to represent each

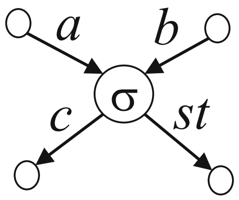


Fig. 5 Example of a statement *st* being φ-enabled. The statement is φ-enabled when $\varphi =$ `[a,c]` $\vee$ `[b,c]` and not φ-enabled when $\varphi =$ `[a,c]`

abstract state as a separate abstract element (an abstract element may be *represented* by another abstract element).

Lines 21–24 check whether a state should be added to the set *states* of explored abstract states, and determine what states need to be added to the workset as a result. First, in Line 21, the algorithm checks whether the state σ' is already *represented* in the set *states* of abstract states. If it is not represented in *states*, Line 22 adds it to the set using the join operation of the underlying abstract domain. This join operation may affect other states that now need to be explored. To determined what abstract states are affected and need to be explored, a set X of abstract states is computed in Line 23. Finally, the states in the set X are added to the workset in Line 24.

In the algorithm, we use the join operator of the abstract domain to add new states to the set *states* of explored abstract states (Line 22) and the workset (Line 24). More generally, the algorithm can use a widening operator [8] when required.

5.2 Enumerating invalid traces

The pseudo-code of Algorithm 1 computes (declaratively) the set of traces invalid Π (Line 3) that are consistent with the accumulated φ. To realize this algorithm, we replace the declarative expression in Line 3 with a call to function `Traces`:

$$\Pi = \texttt{Traces}(Init_P^{\natural}, \{\sigma'\}, states \setminus workset, \varphi)$$

in Algorithm 1, where σ' is a state that violates the specification S (checked in Line 8).

Function *Traces* (X, Y, V, φ)

```
Input: Set of abstract states $X, Y, V$, Atomicity Formula $\varphi$
Output: Set of traces from $X$ to $Y$ passing in $V$, satisfying $\varphi$
$workset = \{t \mid src(t) \in V \setminus X, dst(t) \in Y\}$
$result = \{t \mid src(t) \in X, dst(t) \in Y\}$
while $workset$ is not empty do
    $\pi =$ select and remove interleaving from $workset$
    foreach Statement $st$ and state $\sigma \in V$ such that
    $[\![st]\!]_\beta(\sigma) \sqsubseteq_B src(\pi_0)$ do
        t = transition $(\sigma, src(\pi_0))$ labeled with $st$
        $\pi' = t.\pi$
        if $\pi' \models \varphi$ and $\pi'$ is acyclic then
            if $\sigma \in X$ then $result = result \cup \{\pi'\}$
            else $workset = workset \cup \{\pi'\}$
        end
    end
end
return $result$
```

The function `Traces`(X, Y, V, φ) enumerates all traces that start in a state in $X \subseteq V$, end in a state in $Y \subseteq V$, go only through states in V and satisfy the atomicity constraint φ. It works by performing a backward exploration starting from states in Y and extending interleaving suffixes backwards. A suffix is further extended only as long as it satisfies φ. Thus, the algorithm leverages the constraints that are already accumulated in the atomicity formula φ to prune the interleavings that have to be explored. The use of φ is critical for the practicality of the approach, as shown experimentally in Sect. 7.

5.3 Additional optimizations

Rebuilding parts of the transition system Instead of rebuilding the whole transition system whenever we add a constraint to φ (Line 14), or whenever we refine the abstraction, we can rebuild only the parts of the transition system that depend on the modification. Following approaches such as [12], we can invalidate only the parts of the abstract transition system that may be affected by the refinement, and avoid recomputation of other parts.

Lazy abstraction Algorithm 2 need not maintain the same abstraction across different interleavings. The algorithm can be adapted to use lazy abstraction refinement as in [12]. Instead of maintaining a single homogenous abstraction α for the entire program, we can maintain different abstractions for different parts of the program, and perform lazy refinement.

Simplification of φ Rather than taking the conjunction of constraints as they are accumulated in φ, we preform (propositional) simplification of φ on-the-fly. This is required in practice, as the number of terms added to φ may be large even for small programs.

Multiple solutions The algorithm as described here only yields a single minimal solution. In practice (and in our implementation, described in Sect. 7), it is often desirable to present the user with a range of possible solutions and let the user make her own choice.

6 Correctness and minimality

In this section, we show that Algorithm 1 computes a correct program with smallest atomic sections, assuming the abstraction is fixed. At the end, we discuss the effect of abstraction refinement, and correctness of Algorithm 2.

6.1 Correctness

In this section, we assume that the abstraction is fixed, i.e., *shouldAvoid* in Algorithm 1 always returns *true*. We assume that the termination of fixpoint computation is guaranteed by widening or by the form of the abstract domain (finite ascending chains condition). Recall that the representation of the abstract fixpoint is always finite, whether the abstract domain is finite or infinite.

The following theorem says that a run of the AGS algorithm terminates with either an abort or a valid program.

Theorem 1 (Correctness) *A run of the AGS algorithm terminates with either an abort or returns a program P' such that*
(1) *P' satisfies S under α, i.e., $[\![P']\!]_\alpha \models S$, and*
(2) *P' admits a subset of interleavings of the original program P, i.e., $[\![P']\!]_\alpha \subseteq [\![P]\!]_\alpha$.*

Sketch of Proof: In every iteration, the AGS algorithm eliminates at least one simple path to error state from the abstract transition system. As a result, the abstract transition system may be modified to take into account the updated atomicity formula φ. However, the abstract transition system is always modified in a way that does not introduce any new paths, in particular it has no new paths to error states.

Because the number of simple paths is finite, the `while` loop in the AGS algorithm terminates either by eliminating all simple paths to error, or finding a path to error that has no context switches and thus it cannot be eliminated by our method. In the latter case, the algorithm aborts. In the former case, the set of traces Π is empty. Than is, any choice of atomic predicates that respects the atomicity formula φ yields a program that satisfies S under α. In particular, a minimal assignment Γ used by `implement(P,`φ`)` yields a program $P\mid_\Gamma$ that satisfies S under α.

The AGS algorithm cannot fix a program whose sequential executions do not satisfy S under α. Otherwise, if there is a way to add atomic sections to P such that the result satisfies S under α, then there exists a run of the AGS algorithm that does not abort, and computes a valid program. In the worst case, it makes the program always execute sequentially.

Theorem 2 *If all sequential executions of P satisfy S under α, then there exists a run of the AGS algorithm that does not abort.*

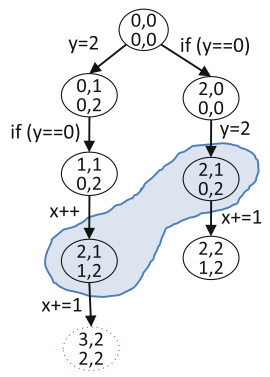
In Algorithm 2, at Line 9, we always choose from `Traces` a trace π that has context switches, if there is one. It guarantees that no run of AGS algorithm aborts if the sequential version of P is valid.

The toy example in Fig. 6a has a single invalid interleaving: `y=2;if(y==0);x++;x+=1`, as shown in Fig. 6b. Because the assertion is checked atomically with `x+=1`, the invalid interleaving `y=2;x+=1;if(y==0);x++` is not possible. However, under parity abstraction, there are two invalid interleavings, due to join (shaded area). One of them is the abstraction of the concrete invalid interleaving, denoted by π_1. The other one is a sequential interleaving, denoted by π_2, in which `T1` executes first, and then `T2`. If the AGS algorithm first chooses to eliminate π_2, it will abort, because there are no context switches to disable. However, if we chose π_1 first, `avoid` will return the atomicity constraint `[y=2,x+=1]`, and the program will be successfully verified under this constraint, using parity abstraction. Similarly, we can construct an example in which a wrong choice leads to larger atomic sections than necessary.

```
 T1 {
0:  if (y==0)
      goto L
1:  x++
2:  L:
 }

 T2 {
0:  y=2
1:  x+=1
    assert x≠y
 }
```

(a)



(b)

Fig. 6 Example demonstrating the effect of join and the choice of different abstract traces to eliminate

6.2 Minimality

Next, we define the notion of a minimally-atomic program, and show how to use the AGS algorithm to compute all minimally-atomic programs for a given input program, specification and abstraction.

Let Γ be a set of atomicity predicates that refer to a program P. In AGS algorithm, we obtain Γ as a satisfying assignment to the atomicity formula φ. Recall from Sect. 4.2 that we realize Γ by adding the tightest atomic sections to P that respect all atomicity predicates in Γ. Let us denote the resulting program by $P\mid_\Gamma$.

Let P' be obtained from P by adding atomic sections. We use $\Gamma(P, P')$ to denote the (unique) set of atomicity predicates that corresponds to these atomic sections.

Minimally atomic programs A valid program is minimally-atomic when removing or shrinking any atomic section in it makes the program invalid.

Definition 1 (*Minimally Atomic*) Consider a program P and an abstraction α. Let P' be a program obtained from P by adding atomic sections. P' is minimally-atomic with respect to α and S if and only if $[\![P']\!]_\alpha \models S$ and for every program P'' obtained from P by adding atomic sections, if $\Gamma(P, P'') \subset \Gamma(P, P')$, then $[\![P'']\!]_\alpha \not\models S$.

The condition $\Gamma(P, P'') \subset \Gamma(P, P')$ means that the atomic sections of P'' is a (strict) subset of those of P'.

We use $MA(P, \alpha, S)$ to denote the set of all minimally-atomic programs with respect to α and S that can be obtained from P.

The programs in $MA(P, \alpha, S)$ have incomparable sets of atomic sections, i.e., for every pair $P', P'' \in MA(P, \alpha, S)$, $\Gamma(P, P') \not\subset \Gamma(P, P'')$. However, they may have the same set of traces under α (and even concrete traces). When the abstraction α is not precise enough to prove that all sequential executions of P satisfy S, $MA(P, \alpha, S)$ is empty. In the rest of this section, we show that every minimally-atomic program can be implemented by AGS algorithm.

Theorem 3 (Minimality) *For every minimally-atomic program $P' \in MA(P, \alpha, S)$, there exists a run of the AGS algorithm that returns P'.*

Sketch of Proof: Let $P' \in MA(P, \alpha, S)$. By Lemma 1, there exists a run of AGS algorithm that terminates with an atomicity formula φ such that $\Gamma(P, P') \models \varphi$. Let Γ be a minimal satisfying assignment to φ such that $\Gamma \subseteq \Gamma(P, P')$. There exists a run of AGS that chooses Γ in `implement` and returns the program $P\,|_\Gamma$.

We show that $P\,|_\Gamma$ is P'. As explained in Sect. 4.2, `implement` ensures that the atomic section of $P\,|_\Gamma$ are the tightest that respect all atomicity predicates in Γ. That is, $\Gamma(P, P\,|_\Gamma)$ is the smallest superset of Γ that can be implemented as atomic sections in P. Because $\Gamma(P, P')$ is also a superset of Γ, $\Gamma(P, P\,|_\Gamma) \subseteq \Gamma(P, P')$. For the sake of contradiction, suppose that $\Gamma(P, P\,|_\Gamma) \subset \Gamma(P, P')$. The correctness of AGS ensures that $[\![P\,|_\Gamma]\!]_\alpha \models S$, and we get a contradiction to the minimality of P' with respect to S and α. Therefore, $\Gamma(P, P\,|_\Gamma) = \Gamma(P, P')$ and $P\,|_\Gamma$ is P'.

Lemma 1 *For every program P' obtained from P by adding atomic sections, if $[\![P']\!]_\alpha \models S$, then there exists a run of the AGS algorithm that terminates with atomicity formula φ such that $\Gamma(P, P') \models \varphi$.*

Sketch of Proof: Let P' such that $[\![P']\!]_\alpha \models S$ and $\Gamma = \Gamma(P, P')$. We show by induction that there exists a run of Algorithm 1 such that $\Gamma \models \varphi$ holds for every loop iteration. In the base case, before entering the loop, φ is $true$ and thus $\Gamma \models \varphi$ trivially holds. Assume that there exists a run such that $\Gamma \models \varphi$ holds at the beginning of some iteration, and show that it holds at the end of the iteration. If the set Π of invalid traces, defined in Line 3 of Algorithm 1, is empty, then the loop terminates and $\Gamma \models \varphi$ hold by inductive hypothesis. Otherwise, the set of traces Π is not empty.

First, we show that Π contains a trace that can be eliminated by adding atomic sections. For the sake of contradiction, suppose that no trace in Π can be eliminated by adding atomic sections. Because Π is not empty, there exits $\Gamma' \models \varphi$ and an invalid trace in $[\![P\,|_{\Gamma'}]\!]_\alpha$ that cannot be eliminated by any assignment of atomic sections. In particular, it cannot be eliminated by Γ, which contradicts the fact that atomic sections of P', captured by Γ, eliminate all invalid traces of $[\![P]\!]_\alpha$.

Let $\pi \in \Pi$ be a trace that can be eliminated, that is, $\texttt{avoid}(\pi) \neq false$. Because $[\![P']\!]_\alpha \models S$ and $\pi \not\models S$, we get that $\pi \notin [\![P']\!]_\alpha$. If α preserves program locations (see Sect. 7), $[\![P']\!]_\alpha = [\![P]\!]_\alpha \cap [\![\Gamma]\!]$. Because $\pi \in [\![P]\!]_\alpha \cap [\![\varphi]\!]$ but $\pi \notin [\![P]\!]_\alpha \cap [\![\Gamma]\!]$, we get that $\pi \notin [\![\Gamma]\!]$. Therefore, using the definition of `avoid` in Sect. 4.1, we get that $\Gamma \models \texttt{avoid}(\pi)$. By inductive hypothesis, $\Gamma \models \varphi$, and we get $\Gamma \models \varphi \wedge \texttt{avoid}(\pi)$, which is the definition of φ at the end of the iteration.

The following proposition is much stronger than Theorem 3, as it requires that a single run of the AGS algorithm yield all minimally-atomic programs. Moreover, the minimally-atomic programs exactly correspond to all minimal satisfying assignments of the atomicity formula φ computed by that run.

Proposition 1 (Minimality-Strong) *If the sequential executions of P satisfy S under α, then there exists a run of the AGS algorithm that yields atomicity formula φ such that*

- *for every minimal satisfying assignment A to φ, the program `implement`$(P, A) \in MA(P, \alpha)$,*
- *for every $P' \in MA(P, \alpha)$, there exists a minimal satisfying assignment A for φ, such that `implement`(P, A) returns P'.*

Correctness and minimality of Algorithm 2 Correctness of the operational version of the AGS algorithm, given in Algorithm 2, follows from the fact that an invalid interleaving eliminated by Algorithm 2 from a partial transition system is also an invalid interleaving that can be chosen by an iteration of Algorithm 1 from the corresponding full transition system. Minimality follows from the fact that the order of (forward) exploration in Algorithm 2 can be chosen to discover error states in a way that exhibits any sequence of minimal invalid interleavings.

Abstraction refinement If refinement is not guaranteed to terminate, then AGS algorithm is not guaranteed to terminate. The reason is that every refinement step may produce new simple invalid interleavings. When the refinement is guaranteed to be monotonic, i.e., abstraction is more precise in every step (e.g., parity to intervals is not monotonic), we can attain minimality under abstraction refinement, by discarding the atomicity constraints φ after each refinement step. When the refinement is not monotonic, we can define a minimally-atomic program to respect any of the explored abstractions. In the case of lazy abstraction, which refines only part of the state-space, the definition of minimality is even more involved.

Finding a minimally-atomic program requires backtracking and it is at least exponential in the size of the abstract transition system of the input program, inline with the known complexity bounds for game-based synthesis [13]. Thus, it is more valuable to invest into a good heuristic. The simple heuristics that we use in the AGS algorithm produce reasonable, and often minimal, synchronization in practice, as we show in the next section.

7 Experience

We built a prototype tool named GUARDIAN based on the AGS algorithm of Sect. 5. We applied GUARDIAN to several interesting programs, inspired by real applications, which we describe next. The abstractions we used are variants of parity and interval domains, where the abstractions differ in what variables are kept relational.

Table 1 summarizes our experimental results. Note that all of our example programs are infinite state, and hence require abstraction for full verification. In our experiments, we were interested in exploring the space of fixes under several abstractions. Even when GUARDIAN found a solution with the original abstraction, we still let it explore solutions with finer abstractions. For every program in the table, we report the number of refinement steps, and number of `avoid` steps performed by the algorithm. In Table 2, we report the atomicity constraints found by GUARDIAN for programs whose code is shown in the paper (atomicity constraints refer to the code).

Table 1 Experimental results

Program	Refinement steps	Avoid steps
Double buffering	1	2
Defragmentation	1	8
3D update	2	23
Array removal	1	17
Array Init	1	56

When using φ-pruning, all experiments ran in less than 10 minutes. Without using φ to restrict exploration, most programs went out of memory exploring a hopelessly large (and redundant) space of interleavings. To enumerate minimal assignments for the atomicity constraints constructed by our algorithm, GUARDIAN uses a model enumerator [31]. In the rest of this section, we describe some of our examples in more detail.

7.1 Abstract domain

In our examples, the abstract domain is a powerset of abstract states. An abstract state s is a tuple $\langle val_s^\natural, pc_s\rangle$, where $val^\natural$ maps program variables to their abstract values, ranging over an abstract domain such as parity, sign or interval.

For $X \subseteq \Sigma_P$, the abstraction function α is defined as follows:

$$\alpha(X) \overset{\text{def}}{=} \sqcup\{\langle\theta(val_s), pc_s\rangle \mid s \in X\}$$

where θ maps concrete values of program variables to their abstract values. We use $\sqcup_\theta$ and $\sqsubseteq_\theta$ to denote the join and order of abstract values. The values of program counters are preserved.

We use the following join and partial order, parametric on the variables tracked relationally. Let $V \subseteq Var$ be a subset of variables. The join $\sqcup$ for the abstract domain A is defined using a relational join over a subset of variables in V and cartesian join over the rest of the variables:

$$s_1 \sqcup s_2 \overset{\text{def}}{=} \begin{cases} \{s_1, s_2\} \text{ if } pc_{s_1} \neq pc_{s_2} \text{ or} \\ \quad \text{exists } v \in V \text{ s.t. } val_{s_1}^\natural(v) \neq val_{s_2}^\natural(v) \\ \{\langle val_{s_1}^\natural \sqcup_\theta val_{s_2}^\natural, pc_{s_1}\rangle\} \text{ otherwise} \end{cases}$$

For $V = Var$ this defines relational join. For $V = \emptyset$ this defines cartesian join. Most of our examples vary the abstraction by varying the relationality in the join. Because the join is parametric on the set V, in the presentation of our examples, we only vary the value of V. The value of V for each example is shown in Table 2. The partial order $\sqsubseteq$ on A is

Table 2 Abstraction and solutions for some of the example programs

Program	Abstraction (set of tracked variables V)	Solution (atomicity constraints)
DBuffer	$\emptyset$	`[fill:L1,fill:L2])` $\vee$ `([render:L1,render:L2]`
	$\{Fill, Render\}$	*true*
Defrag	$\{Barrier, Region, F1, F2, empty\}$	`[D:L1,D:L2]` $\wedge$ `[U:L1,U:L2]` $\wedge$ `[U:L2,U:L3]` $\wedge$ `[U:L3,U:L4]`
	$\{Barrier, Region, F1, F2, empty, i, j\}$	`[D:L1,D:L2]` $\wedge$ `[U:L1,U:L2]`
3D update	$\{x2, x3, y3, z1, z3\}$	`[T1:L2,T1:L3]` $\wedge$ `[T2:L2,T2:L3]` $\wedge$ `[T1:L8,T1:L9]`
	$\{x2, x3, y3, z1, z3, y2, z2\}$	`[T1:L2,T1:L3]` $\wedge$ `[T2:L2,T2:L3])`
	$\{x2, x3, y3, z1, z3, y2, z2, x1, y1\}$	*true*

defined as follows: for all $Y, Y' \subseteq A$, $Y \sqsubseteq Y'$ if and only if for all $s_1 \in Y$ there exists $s_2 \in Y'$ such that $pc_{s_1} = pc_{s_2}$, and $val^{\natural}_{s_1} \sqsubseteq_{\theta} val^{\natural}_{s_2}$.

7.2 Double buffering

This example is motivated by the mechanism of double buffering. Variants of this mechanism are used in a variety of settings, from computer graphics to device drivers. This scheme is illustrated in Fig. 7. There are two buffers of images (`Im`) of length N. The filler thread fills the buffer indexed by the variable `Fill`, while at the same time the rendering thread consumes the buffer indexed by variable `Render`. When the filling completes, the values of the two variables are swaped and the filling restarts. The rendering thread simply renders the image indexed by variable `Render`. To avoid clutter, we assume that rendering is at least twice as fast as filling and hence before `Render` is changed, the value of its buffer has been written to the screen at least once (writing to the screen is idempotent and hence can be repeated).

Specification We would like to prove that the filler and renderer never access the same location simultaneously. Formally:

$$pc(fill) = L2 \wedge pc(render) = L2 \Rightarrow \neg(Fill = Render \wedge i = j)$$

Result Our first solution is obtained with a cartesian parity abstraction. This abstraction loses relationship between variables `Fill`, `Render`, `i` and `j` when states are joined. Formally, the set V of tracked variables is empty ($V = \emptyset$). Recall that the program counters are always kept relational.

With a more refined abstraction, Guardian proves the correctness of the original program. The key reason why we succeeded in this case is that this abstraction maintains the relationship between the values `Fill` and `Render` on each iteration of the loop and can show that these two variables are never equal. In this example, refining the abstraction led to proving the program without any necessary fixes. Further refining the abstraction (e.g. inserting variable i or j in the set V) is not necessary.

```
int Fill = 1;
int Render = 0;
int i = j = 0;
fill() {
 L1:if (i < N) {
 L2:  Im[Fill][i] = read();
 L3:  i += 1;
 L4:  goto L1;
   }
 L5: Fill ^= 1;
 L6: Render ^= 1;
 L7: i = 0;
 L8: goto L1;
}
```

```
render() {
 L1:if (j < N) {
 L2:  write(Im[Render][j]);
 L3:  j += 1;
 L4:  goto L1;
   }

 L5: j = 0;
 L6: goto 1;
}
main() {
 fill() || render();
}
```

Fig. 7 Double buffering

```
Barrier = F1 = F2 = 0;
Region = 2;
Defragment() {
 /* Pick a Region */
 L1: i = Region;
 L2: Region = i - 1;
 L3: empty = i - 2;
 L4: if (i >= N) goto L14;
 /* has free entry? */
 L5: b = Pages[i];
 L6: if (!b  && empty <= 0)
 L7:  empty = i;
 /* Copy Entry */
 L8: if (b  && empty > 0) {
 L9:  Pages[empty] = true;
 L10: empty += 2;
 L11: Pages[i] = false;
   }
 L12: i += 2;
 L13: goto L4;
 /* Barrier Synch */
 L14: Barrier += 1; F1 = 0;
 L15: if (F1 == 1)
    goto L16;
    if (Barrier == 2) {
    Barrier = 0; F2 = 1;
    Region = 2;
    goto L16;
    }
    goto L15;
 L16: goto L1;
}
```

```
Update() {
 /* Pick a Region */
 L1: j = Region;
 L2: Region = j - 1;
 L3: b = Pages[j];
 /* Update the Page */
 L4: if (!b)
    Pages[j] = true;
 /* Barrier Sync */
 L5: Barrier += 1; F2 = 0;
 L6: if (F2 == 1)
    goto L7;
   if (Barrier == 2) {
    Barrier = 0; F1 = 1;
    Region = 2;
    goto L7;
   }
   goto L6;
 L7:
}

main() {
  Defragment() || Update();
}
```

Fig. 8 Concurrent defragmentation

7.3 Concurrent defragmentation

This example is inspired by the problem of defragmentation. Defragmentation algorithms are used in various storage management scenarios (e.g., memory, disk storage) to increase space utilization. In many cases, defragmentation takes place concurrently with an executing application.

In Fig. 8, we show a simplified system where one thread called `Defragment` performs memory compaction concurrently with another thread called `Update` which allocates new entries in memory. The memory is organized as an array of entries called *Pages*. The size of the array N is unknown a-priori. Each entry in the array is either occupied (set to *true*) or free (set to *false*). In practice, an entry may correspond to a heap object or a file on a disk drive. Typically, each entry will also contain various other data fields, which we have omitted here for simplicity.

To avoid synchronization on each entry, the two threads should always work on disjoint regions of memory. To ensure that, at the start of their operation, the two threads handshake and then each picks a separate region to work with (labels L1–L2 in each thread). Note that the handshake is not deterministic, and threads could select different regions on different handshakes. In our case, there are two regions, with the first region containing memory locations with an even index and the second region containing memory locations with an odd index.

`Defragment` works by iterating over the array and moving all used entries to one side of the page. `Update` works by selecting a memory location and updating it if some condition holds.

Specification The threads should always access disjoint memory locations when at the program points accessing shared memory locations. (We omit the specification as it is long and tedious).

Result The resulting constraints are shown in Table 2. The names of the threads have been abbreviated using their first letter. Note that the original program is incorrect (variable *Region* is incremented without any synchronization), and with this more refined abstraction, the inferred correction is not a false positive (e.g. it is not due to an imprecision of the abstraction). However, the constraint `[U:L2,U:L3]` $\wedge$ `[U:L3,U:L4]` inferred with the coarser cartesian abstraction is due to the imprecision of the abstraction.

7.4 Alternating array removal

Our next example shown in Fig. 9 involves two threads removing items from an array in an alternating fashion. The threads are symmetric and hence we use the thread id *tid* to indicate local access. Initially, each thread picks a distinct location to remove (either even or odd). If the item is found at the specified index, it is removed. The two threads then synchronize and subsequently begin removing again, but this time select an item with an alternative index (that is, if the index is even at first, it next becomes odd, and so on). This dynamic exchange of ownership for removed memory locations continues indefinitely. To avoid clutter, we assume that the elements of the array are non-negative and we use the integer value -1 to denote that a memory location has no element residing in it.

```
i[1] = 0; i[2] = 1;
Flag[1] = Flag[2] = 0;
Barrier = 0;

remove(tid, othertid) {
 L1: if (i[tid] == 1)
 L2:  i[tid] = read-even();
 L3: else
 L4:  i[tid] = read-odd();

 L5: v[tid] = read();

 L6: if (A[i[tid]] == v[tid])
 L7:  A[i[tid]] = -1;

 /* Barrier Synchronization */
 L8: Barrier += 1; Flag[tid] = 0;
 L9: if (Flag[tid] == 1)
    goto L10;
   if (Barrier == 2) {
    Barrier = 0; Flag[othertid] = 1;
    goto L10;
   }
   goto L9;

 L10: goto L1;
}

main() {
 remove(1, 2) || remove(2, 1)
}
```

Fig. 9 Alternating array removal

Specification We would like to prove that the two threads never attempt to remove items at the same array index simultaneously. That is, a thread should never try to access a location at labels L6 or L7 when the other thread is writing at L7, and i[1] = i[2].

7.5 3D grid computation

Consider a concurrent program that updates values in a 3 dimensional grid. The program is shown in Fig. 10. Threads `T1` and `T2` should always access disjoint memory locations and hence no synchronization between the threads should be required. `T1` starts by reading a value from the input and then begins a loop which adds this value to the locations on the diagonal of the 2D matrix. We iterate over the diagonal of the 2D (x,y) plane as the value of variable `z1` is fixed to 1 and only `x1` and `y1` change. The loop comprises the statements at labels L2 and L6 in `T1`. After the plane is updated, `T1` updates a value in another plane (L7–L9). For clarity we have only shown the update of a single location, but this can also be extended to update the diagonal. Similarly, thread `T2` updates the diagonal of a 2D plane but this time in the (z,y) dimension. That is, the value of `x2` is fixed and only `y2` and `z2` are updated.

Specification The two threads should never access the same locations simultaneously. That is, if thread `T1` is reading or writing shared data (e.g. at labels L2, L3, L8, L9), `T2` should not be writing simultaneously (e.g. be at L3), and vice versa

```
x1 = 0; y1 = 0; z1 = 1;
x2 = 0; y2 = 1; z2 = 1;
x3 = 0; y3 = 1; z3 = 0;

T1()
 L1: v = read();
 L2: r = A[x1][y1][z1];
 L3: A[x1][y1][z1] = r + v;
 L4: x1 += 1;
 L5: y1 += 1;
 L6: if (x1 < N)
    goto L2;

 L7: v = read();
 L8: r = A[x3][y3][z3];
 L9: A[x3][y3][z3] = r + v;
```

```
T2()
 L1: v = read();
 L2: r = A[x2][y2][z2];
 L3: A[x2][y2][z2] = r + v;
 L4: y2 += 1;
 L5: z2 += 1;
 L6: if (y2 < N)
    goto L2;

main()
 T1() || T2()
```

Fig. 10 Concurrent 3D updating

for `T2`, where the indices of the array being accessed are equal for both threads.

Result As shown in Table 2, refining the abstraction leads to weaker atomicity constraints. In this example, we have three layers of abstractions, each leading to finer-grained solutions.

7.6 Additional examples

Array initialization A three dimensional array is initialized by two threads. The first thread initializes the first two rows of the array, while the second thread initializes the third row. With parity abstraction this program has a fix that makes array accesses atomic. With interval abstraction, GUARDIAN manages to verify the correctness of the original program.

Alternating array removal Elements are removed from a shared array by two threads in an alternating fashion. With a coarser abstraction, atomic sections are required around the array access code. With finer abstraction, GUARDIAN shows that the original program is correct.

8 Related work

Synthesis from temporal specifications Early work by Emerson and Clarke [6] uses temporal specifications to generate a synchronization skeleton. This has been extended by Attie and Emerson to synthesize programs with finer grained atomic sections [1]. Early work by Manna and Wolper [19] synthesizes CSP programs. Pnueli and Rosner [23] consider the problem of synthesizing a reactive module based on an LTL specification. They discuss the problem of *implementability* in this setting, and define necessary and sufficient conditions for the implementability of a given specification. Our work focuses on concurrent programs for shared memory and is based on abstract interpretation, handling infinite-state systems.

Program repair and game-based synthesis Jobstmann et al. [13] consider the problem of *program repair* as a game. In their approach, a game is constructed from (a modified version of) the program to be repaired, and an LTL specification of the correctness property. The problem of repair boils down to finding a winning strategy in that game. This approach has been later extended to provide fault localization and fixing [14,30]. The approach has also been extended to work with predicate abstraction in [11]. In contrast to these, we focus on concurrent programs, use abstract interpretation, and solve the quantitative problem of computing a minimally constrained program.

In [35], we focused on inference of CCR guards in finite-state concurrent programs, where the atomic blocks were not modified. Abstraction-guided synthesis can be viewed as the next general step and addresses the more general problem of infinite-state systems, employs abstract interpretation, and infers atomicity constraints (as opposed to only inferring guards).

Kuperstein et al. [16] present an approach for automatic inference of memory fences in programs running on relaxed memory models. Their approach targets finite-state programs and does not employ abstraction. In a followup work [17], they present partial-coherence abstractions for relaxed memory model that allow fence inference in a framework similar to the AGS algorithm.

Dynamic approaches The problem of restricting the program to valid executions can be addressed by monitoring the program at *runtime* and forcing it to avoid executions that violate the specification. However, restricting the executions of a program at runtime requires a recovery mechanism in case the program already performed a step that violates the specification, and/or a predictive mechanism to check whether future steps lead to a violation.

Existing approaches using recovery mechanisms typically require user annotations to define a corrective action to be taken when the specification is violated. For example, software transactional memory [26] is a special case of a recovery mechanism in which the user provides atomicity annotations defining atomic sections. The system then requires the absence of read/write conflicts, and if this property is violated, the execution of an atomic section is restarted. Other examples include Tolerace [22] which creates local copies of variables to detect and recover from races, and ISOLATOR [24] which can recover from violations of isolation.

Search-based synthesis In previous work [33,34], we used a semi-automated approach for exploring a space concurrent garbage collectors and linearizable data-structures. The work used a search procedure and an abstraction specifically geared towards the safety property required for the specific domain.

In *sketching* [28,29], the user provides a reference program of the desired implementation and some sketches which partially specify certain optimized functions. The sketching compiler automatically fills in the missing low-level details to create an optimized implementation. Sketching has been used for bounded programs and in special cases of unbounded domains [27]. In [28], finding a candidate solution is done using a counterexample-guided inductive synthesis (CEGIS) algorithm that uses a backing bounded-checking procedure. Candidates are generated iteratively and run through the checker. Counterexamples are used to limit the next candidates to be generated. In contrast, rather than generating candidates and checking them, in our approach, the synthesizer is part of the verification procedure and is based on abstract interpretation. Further, in contrast to sketching, which aims to find *some* solution for the sketch, we are interested in finding a solution with minimal synchronization.

Locks for Atomicity There have been several works on inferring locks for atomic sections. McCloskey et al. [20] present a tool called Autolocker. The tool takes as input a program that has been manually annotated with (i) atomic sections and (ii) a mapping between locks and memory locations protected by these locks. Autolocker produces a program that implements the atomic sections in (i) with the locks in (ii). Further work by Emmi et al. [9] proposed a technique for automating part (ii). The assignment of locations to locks is solved as an optimization problem where the goal is to minimize the total number of locks while still achieving minimum interference between the computed locks. Cherem et al. [5] propose another alternative for automating (ii) while also computing actual lock placement in the code. Gueta et al. [10] present an approach for automatically realizing atomic sections using fine-grained locks in programs where the shape of the heap is restricted to a forest. Our work is complementary to these approaches, as our focus is not on optimizing the implementation of atomic sections, but on inferring minimally atomic synchronization.

9 Conclusions and future work

We presented a novel algorithm for the automatic synthesis of efficient synchronization in concurrent infinite-state programs (AGS). The synchronization can be realized by modifying either the program or the scheduler. Our algorithm is based on abstract interpretation and thus applies to concurrent infinite-state programs.

The AGS algorithm leads to a new verification approach: it allows for both the abstraction *and* the program to be modified simultaneously until the abstraction is precise enough to verify the (modified) program. This enables verification of a program in cases where it would have otherwise failed.

We implemented the AGS approach in a prototype tool named Guardian, and successfully applied it to several small concurrent programs. Guardian works with various numerical abstractions. In the future, we intend to investigate its use with finer abstract domains, such as the trace partitioning domain [25], which is a natural fit for our setting, as it allows to abstract interleavings with varying degrees of precision.

We demonstrated our approach using atomic sections as the synchronization primitive, but `avoid` and `implement` can be realized using other synchronization primitives. In the future, we intend to explore extensions of AGS to other synchronization primitives.

The general AGS algorithm described in this paper can also be applied in a dynamic setting, where invalid interleavings are obtained by running the program driven by test-cases (e.g., [18]). In such a setting, the constraints obtained from dynamic executions can be used to give the user partial program corrections, or used to limit the space that has to be explored statically.

Acknowledgments The authors wish to thank Mooly Sagiv for many insightful comments on an earlier version of this work. This research was partially supported by The Israeli Science Foundation (grant no. 965/10). Part of the research was done while the authors were with the IBM T.J. Watson Research Center.

References

1. Attie, P., Emerson, E.: Synthesis of concurrent systems for an atomic read/atomic write model of computation, pp. 111–120. In: PODC '96, ACM, Berlin (1996)
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN, pp. 103–122 (2001)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207 (2003)
4. Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: CAV, pp. 140–156 (2009)
5. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI, pp. 304–315 (2008)
6. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, pp. 52–71 (1982)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV, pp. 154–169 (2000)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL, pp. 238–252 (1977)
9. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL, pp. 291–296 (2007)
10. Golan-Gueta, G., Bronson, N., Aiken, A., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic fine-grain locking using shape properties. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 225–242. OOPSLA '11. ACM, New York (2011)
11. Griesmayer, A., Bloem, R.P., Cook, B.: Repair of boolean programs with an application to C. In: CAV, pp. 358–371 (2006)
12. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
13. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: CAV, pp. 226–238 (2005)
14. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. (2008)
15. Kuperstein, M.: Preserving correctness under relaxed memory models. Master's thesis, Technion (2012)
16. Kuperstein, M., Vechev, M., Yahav, E.: Automatic fence inference. In: FMCAD'10: Formal Methods in Computer Aided Design (2010)
17. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, pp. 187–198. ACM, New York (2011)

18. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI'12: Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (2012)
19. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. **6**(1), 68–93 (1984)
20. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: POPL, pp. 346–358 (2006)
21. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. **19**(1), 31–100 (2006)
22. Nagpaly, R., Pattabiramanz, K., Kirovski, D., Zorn, B.: Tolerace: Tolerating and detecting races. In: STMCS: Second Workshop on Software Tools for Multi-Core Systems (2007)
23. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL '89, pp. 179–190. ACM, New York (1989)
24. Rajamani, S., Ramalingam, G., Ranganath, V.-P., Vaswani, K.: Controlling non-determinism for semantic guarantees. In: Exploiting Concurrency Efficiently and Correctly—(EC)2 (2008)
25. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26 (2007)
26. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95, pp. 204–213. ACM, New York (1995)
27. Solar-Lezama, A., Arnold, G., Tancau, L., Bodík, R., Saraswat, V.A., Seshia, S.A.: Sketching stencils. In: PLDI, pp. 167–178 (2007)
28. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
29. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by Sketching for Bit-Streaming Programs. In: PLDI, pp. 281–294 (2005)
30. Staber, S., Jobstmann, B., Bloem, R.: Finding and fixing faults. In: CHARME, pp. 35–49 (2005)
31. The SAT4J SAT solver. http://www.sat4j.org/
32. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the Symposium on Logic in Computer Science, pp. 332–344 (1986)
33. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI, pp. 125–135 (2008)
34. Vechev, M.T., Yahav, E., Bacon, D.F., Rinetzky, N.: Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In: PLDI, pp. 456–467 (2007)
35. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: TACAS, pp. 139–154 (2009)