Purifying Causal Atomicity

Benjamin S. Lerner and Dan Grossman

University of Washington
{blerner, djg}@cs.washington.edu

Abstract. Atomicity-checking is a powerful approach for finding subtle concurrency errors in shared-memory multithreaded code. The goal is to verify that certain code sections appear to execute atomically to all other threads. This paper extends Farzan and Madhusudan's recent work on *causal atomicity* [1], which uses a translation to Petri nets to avoid much of the imprecision of type-system based approaches, to support *purity annotations* in the style of Flanagan et al. [2]. Purity avoids imprecision for several key idioms, but it has previously been used only in the type-system setting. Our work is (1) *compositional*: a different purity analysis could be implemented with minimal extra effort, and similarly another atomicity criterion could be checked without changing the purity analysis, and (2) a *conservative extension*: the analysis of any program that does not use purity annotations is equivalent to the original analysis.

1 Introduction

Static analysis of lock-based shared-memory multithreaded programs is a valuable tool for finding programming errors or verifying their absence. An important recent trend is toward analyzing higher-level concurrency properties. In particular, instead of detecting data races (e.g., a write to a thread-shared variable not protected by a lock), we can verify that an entire code block is *atomic*: it appears to happen either all-at-once or not-at-all to any other thread. Atomicity is a common requirement for code blocks, and the absence of data races is neither necessary nor sufficient for atomicity.

Atomicity checking takes a multithreaded program with certain code sections annotated that they should be atomic, which we write `atomic` $\{ s \}$, and verifies that s uses mechanisms such as locks correctly to achieve atomicity. Prior work on static analysis for atomicity checking has used either type-and-effect systems or model-checking techniques. Reachability queries over Petri nets, which our work uses, represent a recent effort in the latter style.

The type-system approach [2–6] uses syntax-directed rules to assign each program statement an atomicity based on Lipton's theory of movers [7]. Though efficient, elegant, and relatively easy to prove correct, type systems are susceptible to false positives (over-approximations) resulting from (1) the syntactic structure of the code, and (2) the thread-modular assumption that any other code in the program might run in parallel with any atomic section. Model-checking approaches [8–11] can improve precision by modeling the whole program and tracking inter-thread dependencies through shared variables and locks. Using Petri

nets to model programs is particularly convenient because data- and control-dependencies are modeled directly and atomicity checking can be formulated as a query over the net's state-space that existing tools can process.

Our work extends and adapts prior Petri-net work [1] to support *purity* annotations, which previously have been investigated only via type systems [2]. A pure block `pure` { s } must either do no writes or terminate "abruptly" by executing a `break` statement. As later examples will demonstrate, pure blocks let us revise the definition of atomicity to allow several correct coding idioms that otherwise would be considered non-atomic (i.e., atomicity-checking false positives). However, a sound analysis must ensure pure blocks are, indeed, pure, meaning they terminate abruptly or have no effect.

Our overall contribution is an atomicity checker supporting purity annotations using Petri nets. This approach avoids the false positives from type systems and the false positives from idioms requiring purity. We have rigorously defined our analysis for a small core programming language, implemented it (using CPN-Tools [12] for building and querying Petri nets), and checked many examples including all examples in this paper. More specifically, our work has produced the following insights, results, and contributions:

- We show that the Petri-net model of causal atomicity is strictly more powerful than the type-system model of reducible atomicity. That is, every program that type-checks under the system in [13] can pass as causally atomic under the model in [1]. To our knowledge, this is the first formal treatment relating the expressiveness of a type-system approach for atomicity checking to a model-checking approach.
- We show how purity-checking can be encoded in a Petri net using several key technical insights, such as maintaining lock-sets in thread-local storage and using colored markings to track whether a pure block has done a write.
- We show that a single Petri net can compute both purity- and atomicity-checking in a way that is a *conservative extension* of the atomicity analysis (if a program has no pure-blocks, the checker is equivalent to prior causal-atomicity checkers) and *compositional* (purity- and atomicity-checking are essentially orthogonal, allowing variants of each to be developed independently). Moreover, we show the combined analysis is more precise than the union of the two analyses.

Space constraints compel only a high-level overview of our analysis, focusing on how purity requires several novel extensions over the closely related work of Farzan and Madhusudan [1]. A companion technical report [14] contains formal definitions and proofs. Our implementation, including the full translation from programs to Petri nets, is also available [15]. Section 2 explains the benefits of purity and Petri nets via examples. Section 3 introduces our core language and the basics of Petri nets. Section 4 defines our translation from programs to Petri nets and an atomicity-checker over the resulting net. Section 5 briefly describes our implementation. Section 6 formally describes how our approach is more powerful than type systems with purity annotations.

2 Atomicity Analyses by Example

2.1 Reducible atomicity via type systems

Lipton's theory of movers [7] categorizes statements by how they can be reordered relative to other threads' statements without affecting the final result. For example, a lock acquisition commutes with a subsequent operation in another thread, as no other thread can use the lock just acquired; we say acquisitions are *right* movers. Classifying variable accesses depends on their race-freedom: if no race exists, the access can commute in *both* directions; if there is a race it cannot commute. Sequences of statements with zero or more right-movers followed by at most one non-mover followed by zero or more left-movers always can be reduced to a serial execution; this property is called *reducible atomicity*.

```
atomic {
  t = balance;
  acquire(m);
  if (t > 10)
    then balance := 0
    else balance := t-10;
  release(m);
}
```

Fig. 1. Bank withdrawal

Flanagan and Qadeer [3] use a type-and-effect system to define a static analysis that checks for reducible atomicity. For example, for the code in Figure 1, where `t` is a local variable and `balance` is a global variable normally protected by lock `m`, the type system can determine the atomic-block is incorrect. Swapping the first two statements fixes the error and atomicity-checking succeeds.

Reducible atomicity has two key limitations later examples demonstrate. First, type-checking follows the syntactic structure of the code, which leads to brittle results: clearly equivalent programs can differ in whether atomicity-checking succeeds. Second, atomicity-checking one thread does not examine the code in other threads to determine more precisely what effects might be observed.

2.2 Causal atomicity via Petri nets

One limitation of mover-based systems comes from over-approximating the effect of variable accesses. In Figure 1, a race condition exists where another thread could change `balance` between the two accesses; this race makes atomicity checking fail. However, if some program invariant prevents the race condition (sometimes called "higher-order locking"), then the code is actually atomic.

This weakness is the strength of the Petri-net approach of *causal atomicity* [1]. By examining the state space of whole-program behaviors, it can use additional context to see if the code in Figure 1 is atomic. As an extreme example, checking causal atomicity always succeeds if there is only one thread. More realistically, it can detect that the program in Figure 2 is causally atomic, despite the data races on `X` and `Y`. As Section 4 explains, by translating the program to an appropriate Petri net, we can check that neither one of the other threads can tell that it observed an intermediate state of the atomic block.

Thread 1	Thread 2	Thread 3

`s := X`	`atomic {`
	`  X := 1;`
	`  Y := 2`
	`}`
		`t := Y`

Fig. 2. Causally but not reducibly atomic

2.3 Pure-reducible atomicity

Several well-known idioms, such as double-checked locking and waiting on condition variables, are not reducibly atomic.[1] But no intermediate states of these idioms are observable, and their behavior in all cases is indistinguishable from cases where they are indeed reducibly atomic. For these idioms, this more "abstract" approach to atomicity suffices to show program correctness; conversely, if a correctness property fails under this broader notion, it also fails under reducible atomicity. We can make this observation precise using the notion of *purity*.

In Figure 3, the variable `x` is read without holding the protecting lock `l`, a potential race condition and therefore an "atomic" operation using reducible atomicity. The critical section guarded by `l` is also an "atomic" operation; the sequence of two atomic operations is not atomic. However, consider all ways this code can actually run: If the first `if`-test succeeds, the code `break`s and skips the remaining code; this code path is indeed atomic. Otherwise, the `if`-statement does nothing and is followed by the critical section. Crucially, in this latter case, the entire `pure` block modifies no state. So if control reaches the `pure` block's end, the block is equivalent to a no-op, and a no-op followed by an atomic operation is atomic. Flanagan et al. [2] generalized this observation into the notion of *pure-reducible atomicity* (the authors used the term "abstract atomicity"): If a `pure` block always either breaks or modifies no state, then atomicity checking can treat it as a no-op. (Note: Without the guard protecting `x` := `newX`, the code still is pure-reducibly atomic, though it is not atomic when run. As noted above, pure-reducible atomicity guarantees correctness only when the abstract and concrete behaviors of the program coincide, which is not true here.)

```
atomic { block {
  pure { if (x != null) then { break } };
  acquire(l);
  if (x == null) then { x := newX };
  release(l);
}}
```

Fig. 3. Double-checked locking: pure-reducibly atomic, but not reducible

In the example above, the purity annotation ensures that `pure` code changes no shared state. In general, it must ensure also that no locks are changed (no initially-held locks are left released, nor initially-unheld locks acquired). In Figure 4, the code acquires lock `l` and `wait`s until `x` becomes false; we model `wait` as a `release`/`acquire` pair. Such code is never reducibly atomic. But each loop iteration is *pure*, and every execution of the `atomic` block is equivalent to one where the loop condition is false even before the block begins: such an execution acquires the lock, skips the loop, executes the body and releases the lock, which *is* an atomic sequence (assuming ***`body`*** is atomic). Unfortunately, the code must be rewritten to accomodate the syntactic restrictions of the type-system before it will validate as pure-reducibly atomic. As the authors noted, not all uses of `wait` can be so reorganized, even when such uses should be pure-reducibly atomic.

```
atomic {
  acquire(l);
  pure { while (x) {
    release(l); acquire(l)
  }}
  ...body...
  release(l);
}
```

Fig. 4. Simple `wait` loop

[1] It is well known that the double-checked locking idiom is incorrect under many relaxed memory-consistency models [16]; we assume sequential consistency here.

Thread 1	Thread 2	Thread 3	Thread 4

`s := X`	`atomic {`
	`X := 1;`
	`pure { while (Z != 5) skip };`
	`Y := 2`
	`}`
		`t := Y`	`Z := 5`

Fig. 5. Pure-causally atomic, but not reducibly, pure-reducibly or causally atomic

2.4 Pure-causal atomicity

In the rest of this paper, we show that combining the advantages of pure- reducible atomicity and causal atomicity yields a system that can validate all the above examples as well as examples no previous work can, under a definition we call *pure-causal atomicity*.

In the same way causal atomicity can validate programs more precisely than reducible atomicity, so pure-causal atomicity can validate programs more precisely than pure-reducible atomicity. The program in Figure 5 highlights these differences. Looking at the first three threads, and ignoring the access to `Z` for a moment, we see the same example as in Figure 2, so we know this cannot be pure-reducibly atomic. Since it has a loop that may repeatedly access `Z`, a shared variable modified in the fourth thread, we know this cannot be causally atomic. Yet this is a realistic scenario: one producer thread (thread 4, above); two consumer threads (threads 1 and 3); and a thread which produces some output, waits for an input, and produces more output (thread 2). Under our system, this code does validate as pure-causally atomic: all iterations through the loop are pure, and hence can be skipped.

3 Preliminaries

3.1 The General Approach

We explain our checker for pure-causal atomicity in stages. We present the core language in Section 3.2, and the essential concepts of Petri nets in Section 3.3. Our approach is based heavily on causal atomicity [1], so we begin by explaining that system's design. Causal atomicity first inductively translates programs from the source language into a Petri net that models the control flow and inter-thread contention over shared variables and locks, and abstracts other details (such as the values of variables). The precise notion of causal atomicity is then expressible as a decidable property of traces over that Petri net. Finally, this property can be directly computed using *colored reachability*, a standard analysis over Petri nets supported by existing tools. We present the translation of programs to Petri nets in Section 4.1, the definition of causal atomicity in Section 4.2, and the coloring rules in Section 4.3.

Our notion of pure-causal atomicity extends each of these three stages. First, we extend the translation function to support `pure` annotations (Section 4.4). We then refine the key property over traces to incorporate the results of the

$$
\begin{array}{rrcl}
P \in & \mathit{Prog} & ::= & \cdot \mid T || P \\
T \in & \mathit{Thread} & ::= & s \\
s \in & \mathit{Stmt} & ::= & x := e \mid s \,;\, s \mid \texttt{if}\ e\ s\ s \mid \texttt{loop}\ s \mid \texttt{atomic}\, s \mid \texttt{skip} \\
& & \mid & \texttt{acquire}\ l \mid \texttt{release}\ l \mid \texttt{block}\ s \mid \texttt{break} \mid \texttt{pure}\ s \\
e \in & \mathit{Exp} & ::= & c \mid x \mid p(\bar{e}) \\
x \in & \mathit{Var} & & \\
c \in & \mathit{Const} & = & \mathbb{Z} \uplus \mathbb{B} \\
p \in & \mathit{Prim} & = & \mathit{ArithPrim} \uplus \mathit{LogicPrim} \\
l \in & \mathit{Lock} & &
\end{array}
$$

Fig. 6. Syntax of our language

purity analysis (Section 4.5). Finally, we extend the coloring rules to implement this refined definition (Section 4.6).

3.2 The language

Figure 6 defines the syntax of our language. Most of the statement forms (`skip`, `if`, etc.) have standard semantics. A `loop` repeats infinitely, and requires a `break` statement to abruptly exit by jumping to the end of the nearest enclosing `block` statement. We define $\texttt{while}\,(e)\,\{\,s\,\} \stackrel{\text{def}}{=} \texttt{block loop}\,\{\,\texttt{if}\,(e)\ \texttt{then}\ s\ \texttt{else break}\,\}$.

A program is a fixed number of threads, which we denote by T to distinguish them from substatements. The language also has two hooks for our analyses: a $\texttt{pure}\,\{\,s\,\}$ statement indicates s should be pure, while an $\texttt{atomic}\,\{\,s\,\}$ statement indicates s should be atomic. Both annotations have no runtime effect but are verified statically. Purity requires that on normal termination a code block not perform any variable writes or leave any locks modified (i.e., an unmatched acquire or release).

The semantics of `break` statements is to terminate the current `block`. As such, using `break` within a `pure` block transfers control outside the block; such "abruptly" exiting paths are not checked for purity, thereby permitting `pure` blocks to cause side-effects on abrupt termination (as in the desugaring of the `while` loop in Figure 5 above), and are key to purity's utility.

3.3 Petri Nets

A Petri net [17] is a triple $N = (P, T, F)$, with P a set of *places*, T a set of *transitions*, and F a *flow relation* $F \subseteq (P \times T) \cup (T \times P)$. For our purposes, transitions model instructions in a program, and places model resources such as variables, locks, or the current program position within each thread.

Places can be marked with *tokens*, which are drawn from a finite set of colors. The assignment of tokens to places is called a *marking*. We will restrict ourselves to nets where in all reachable markings each place has at most one token. A transition t is enabled when all its pre-conditions (all places p for which an arc

(p, t) exists) are marked. When enabled transitions fire, they remove the tokens on their pre-conditions and place new, possibly differently-colored, tokens on their post-conditions (all places p with an arc (t, p)). Starting from some initial marking M_0, a sequence of transitions $t_1, t_2, \ldots$ is called a *firing sequence* if transition t_i is enabled in marking M_{i-1} and, after firing, produces marking M_i; each marking M_i is *reachable* from M_0. For our purposes, as transitions correspond to instructions in the program, firing sequences correspond to execution schedules of the program. Moreover, a marking summarizes the state of the program: the current program positions for each thread, the sets of locks currently held, and various state properties on variables. An example Petri net constructed by our analysis is drawn in Figure 8 on page 9; not shown is the marking, which includes the variable places and one program counter at some point in each thread.

The *neighborhood* of a transition is defined as the union of its pre- and post-conditions. If the neighborhoods of two transitions t_1, t_2 are disjoint, the transitions are said to be *independent*, denoted $t_1 I t_2$. Intuitively, if $t_1 I t_2$, the firing of one transition cannot immediately influence the firing of the other, and once both transitions have fired consecutively, the marking of the net is guaranteed to be the same regardless of the firing order. All transitions that are not independent are called dependent, denoted $t_1 D t_2$. For our purposes, as places correspond to variables, locks and program positions, dependencies between transitions correspond to control- and data-dependencies in the program. It is this notion of dependence that gives rise to the definition of causal atomicity. Given a firing sequence, the dependence relation captures exactly which scheduling interactions between threads matter, and which are artifacts of the current schedule.

To abstract away from firing sequences and capture the dependence notion directly, define a *trace* of a Petri net as a triple $(\mathcal{E}, \preceq, \lambda)$, where $\mathcal{E}$ is a set of *events* corresponding to the firing of transitions, $\lambda : \mathcal{E} \to T$ *labels* an event with the transition that fired, and $\preceq$ is a partial order on events that respects the dependence relation. Specifically, if $\lambda(e_1) D \lambda(e_2)$, then $e_1 \preceq e_2 \vee e_2 \preceq e_1$, and if $e_1 \prec\!\!\cdot\, e_2$, then $\lambda(e_1) D \lambda(e_2)$, where $e_1 \prec\!\!\cdot\, e_2 \stackrel{\text{def}}{=} e_1 \preceq e_2 \wedge \nexists e. e_1 \prec e \prec e_2$. These definitions imply that firing sequences are *linearizations* of traces; they are one particular ordering that respects dependencies. Furthermore, all firing sequences corresponding to the same trace lead to the same marking (i.e., program state).

4 Atomicity checking via Petri nets

4.1 Causal atomicity: Building the net

Figure 7 shows part of the translation function TRANS(s) from program statements into Petri nets. Circles represent places, boxes represent transitions, and arrows represent the flow relation (ignore for now the dashed lines). “Circle-boxes” represent subnets of the Petri net generated by recursive calls to TRANS, and depict a key structural property of the translation: any statement or expression yields a subnet beginning with a unique place p_{in} and ending with some number of arcs out of some number of transitions; this structure is inductively maintained by the translation. Our translation is value-insensitive. For

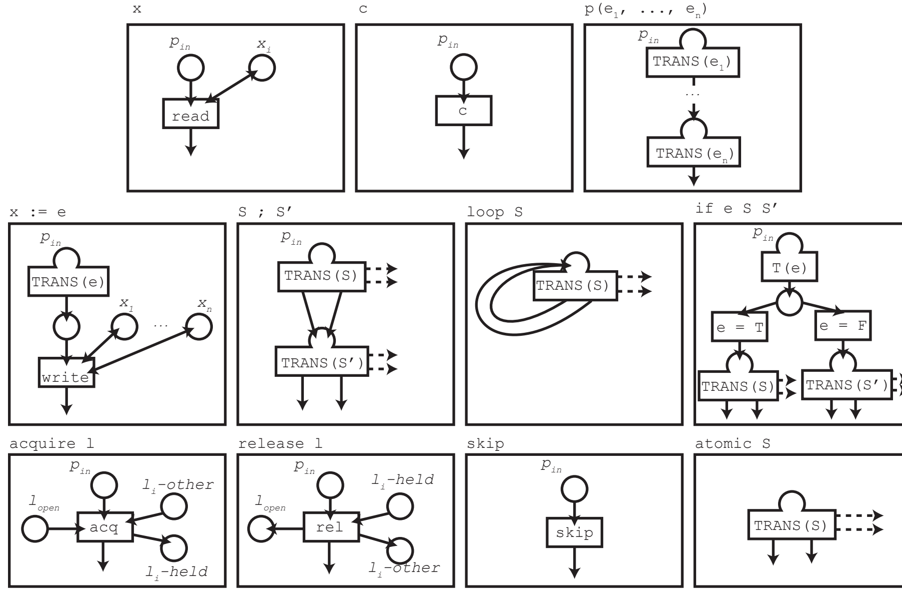

Fig. 7. The TRANS function for basic constructs in our language. Dashed lines and the l_i*-held* and l_i*-other* places will be used later.

example, TRANS(`if` (e) `then` S `else` S') first translates the guard expression, TRANS(e). The two transitions $e = T$ and $e = F$ are thereby enabled, and precisely one of them, chosen nondeterministically, fires, at which point the token is passed into TRANS(S) or TRANS(S'). The outgoing arcs of the `if` statement are the union of the outgoing arcs of both branches.

While translating statements encodes the control flow, we must also encode all resources—variables and locks—that the program uses. For each variable v we construct an array of variable places v_i for each of the n threads $1 \leq i \leq n$, each marked with a single token. Crucially, reading a variable v in thread i depends only on the place v_i, while writing to v depends on every place $v_1, \ldots, v_n$. This ensures all writes are causally related and read-write conflicts are faithfully reproduced, while multiple read events are causally independent. To model lock operations, a single place l_{open} is produced for every lock l, marked with a single token. Acquiring a lock removes the token, while releasing a lock replaces it. The natural behavior of the net therefore models the mutual exclusion of locks—see the TRANS case for lock acquisition. Later we will revise this encoding slightly to support the needs of purity checking.

The complete translation for a given program $P = T_1 || \cdots || T_n$, then, translates each thread, the variable places and the lock places as above, and adds an `ERROR` place to be described below. We mark every variable place, every lock place, and the entry point of each thread to initialize the net. An example of this translation, showing the (unmarked) net corresponding to the program in Figure 2, is shown in Figure 8.

4.2 Defining causal atomicity

The essence of atomicity is that all instructions in an atomic block must appear to happen indivisibly to all other threads. As such, given a trace with two events in an atomic block in some thread T (i.e., events whose labels are transitions in TRANS(T)), no event in some second thread T' (i.e., events labeled by transitions in TRANS(T')) should be causally "between" the two. Such events would show a data dependence (or antidependence) flowing out of and back into the supposedly atomic block, violating its atomicity.

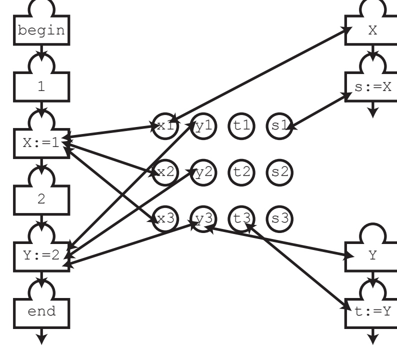

Fig. 8. Translation of Figure 2

Formally, let e^T denote that event e occurs in thread T, and let S be the subnet from translating some `atomic` block in a program P. Then S is *causally atomic*[2] if there does not exist a trace where

$$\exists e_1^T, e_2^{T'}, e_3^T \in \mathcal{E} \,.\, e_1 \in \mathrm{START}(S) \wedge e_1 \preceq e_2 \preceq e_3 \wedge \nexists e^T \in \mathrm{END}(S) \,.\, e_1 \preceq e \preceq e_3$$

where START(S) and END(S) are the sets of events labeled by the first and last transitions that can possibly fire in S, and T and T' are distinct threads.

4.3 Computing causal atomicity: coloring the net

Rather than check for causal atomicity violations by generating and examining an infinite number of traces (which is not an algorithm), we examine the *state space* of possible markings of the net. Specifically, we can use *colored tokens* to encode our definition to see if causality flows across threads in an unacceptable way, and query whether a particular (bad) coloring is possible. We simply give each mark a color drawn from the set $\{A, B, Y, R\}$, and change colors as transitions fire:

- Initially, all tokens are colored A (achromatic).
- On entry to an `atomic` block in thread T, the mark may be turned B (blue). This corresponds to event e_1^T above, and means we guess this block may in fact be non-atomic. Any transition in thread t with any B inputs will propagate B to all outputs.
- If a transition in another thread T' has a B input, it will turn all outputs Y (yellow); this corresponds to event $e_2^{T'}$. Any transition not in thread T with any Y inputs will propagate Y to all outputs.
- Finally, if a transition in thread T sees a Y input, it turns the mark R (red); this is the final event e_3^T, denoting an atomicity violation. If the end of the atomic block is reached (event e^T) and the color is not R, this trace does not show an atomicity violation (we guessed wrong turning the token B).

[2] This is a slightly different formulation than the definition in [1], but we prove the two equivalent in our companion technical report [14].

If there is no trace where the token is turned B at the start of some `atomic` block and turns R by the end, then the program is causally atomic. This algorithm will always terminate, as our state space—the set of reachable markings—is finite (recall we constructed our net such that markings are simply subsets of the net's places), and there exist efficient algorithms to answer these reachability queries lazily, without computing the entire state space.

To see these rules in action, consider how they might apply to the net in Figure 8. Initially all marks start uncolored; when the `begin` transition fires, the mark in thread 2 turns blue. When thread 2 executes `X` := `1`, the place `x1` is marked blue as well. Suppose that thread 1 now starts and fires `X`; its token (and that of `x1`) would turn yellow. However, there is no other transition in thread 2 that will access `x1`, so there is no way for the yellow token to turn red. Similar arguments can be made for other execution orders.

4.4 Pure-causal atomicity: Enhancements for purity

The translation described so far checks causal atomicity, but contains no way to check for or exploit purity in the program. We now show how to encode a purity analysis as a query over Petri nets, using its results to improve the atomicity queries, and moreover querying both purity and atomicity with the same net.

Figure 9 depicts the rest of the TRANS function. First, for each lock l, `acquire` and `release` operations now not only access a global place l_{open}, but also shuttle a mark between two thread-local places, l_i-*held* (only marked when lock l is held by thread i) and l_i-*other* (marked otherwise). These two places permit a thread to examine the set of locks it currently holds, *without any causal dependencies on other threads* (i.e., without accessing any "thread-shared" places); this property will be crucial for purity checking. Next, `block` statements define a target for the non-local jump behavior of `break` statements; this is depicted in our diagrams by the horizontal dashed arrow of `break` (and those of all inductive calls to TRANS). All `block`s need to provide is a place for those arcs to target; this place then outputs via a normally-terminating arc.

Almost all the increased complexity comes from the translation of `pure` statements. We need to ensure that:

- On any control-flow path that reaches the end of the pure block, the thread does no writes, any locks released are (re)acquired, and any locks acquired are released. Otherwise a purity violation is reported.
- On any control-flow path that reaches a `break` statement before the end of the `pure` block, any causal-atomicity checking (for an `atomic` block in this or any thread) is performed as usual, possibly reporting an atomicity violation.
- On any control-flow path that reaches the end of the `pure` block, the `pure` block's actions must *not* lead to an atomicity violation. These are exactly the false positives we avoid via purity annotations.

To encode these three issues, entry to TRANS(`pure` s) makes a nondeterministic 3-way choice. If the `purity` transition fires, the subsequent execution of the

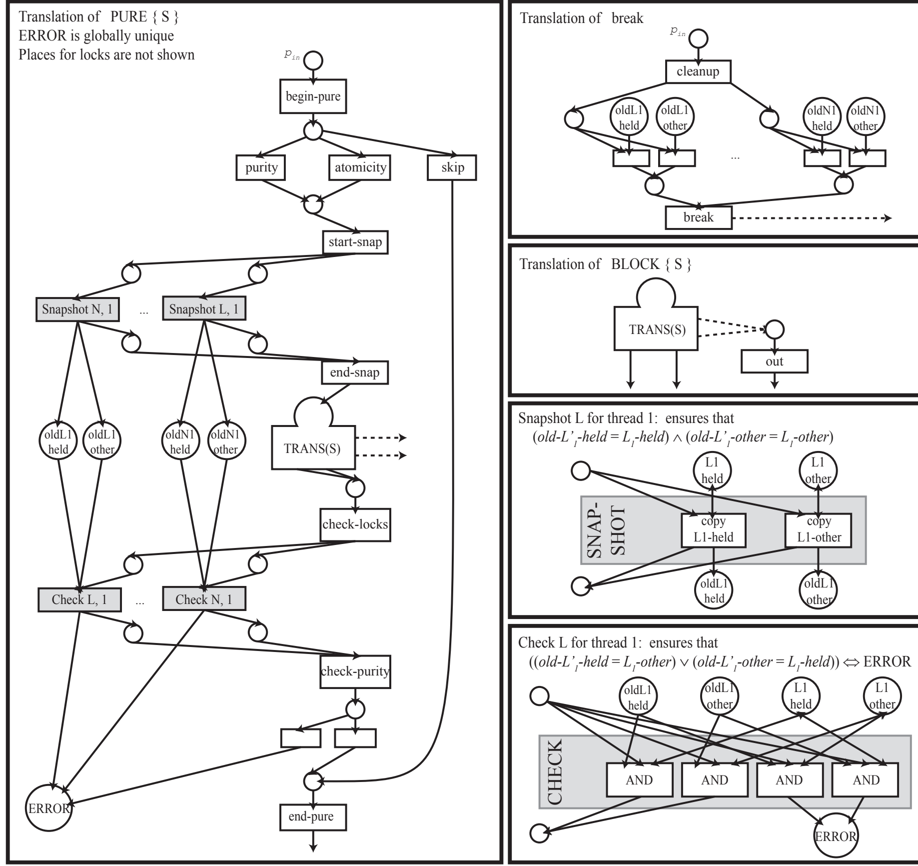

Fig. 9. The rest of the TRANS construction to implement purity checking of locksets and variable accesses. Each lock place L and L_i-*other* starts off marked.

block will fail if the block is not pure (succeeding "vacuously" if a `break` statement is reached) and no atomicity checking is done. If the `atomicity` transition fires, the subsequent execution of the block will do normal atomicity checking (succeeding "vacuously" if the end of the block is reached *without* executing a `break` statement). If the `skip` transition fires, control transfers immediately to the end of the `pure` block.

The `skip` option is what allows the `purity` option to maintain no information with regard to atomicity checking. Since the `purity` option ensures no path to the end of the pure block can affect or be affected by another thread, no transitions that occur along such a path can themselves lead to a violation of causal atomicity. However, we cannot completely ignore control paths that happen to include `pure` blocks that do not execute `break` statements. For example, code of the form `x := 1; pure{...}; y := x` might lead to an atomicity violation; the `skip` option covers such cases.

Let us now focus on how we actually check for purity when the `purity` transition is fired. To handle variable mutations, we initialize the color of the token to a "known good" state, and update it to record a "potential problem" once a write occurs. On exiting a pure block normally, only the known-good color can continue, while the other leads to `ERROR`. If the block `break`s, however, this color check is bypassed, matching our definition that abrupt termination can mutate state. More formally, the color checks ensure that all mutations are post-dominated by a `break` statement.

To handle lock manipulations, we take a "snapshot" of the set of held locks just before executing the body of the `pure` block, and "check" that the same locks are held on normal termination. Two points are crucial to the correctness of these constructions: first, we are guaranteed that precisely one token is present on either l_i-*held* or l_i-*other*, and therefore precisely one transition is enabled in the `snapshot` construction. The snapshot maintains this property for its two output places, *old*-l_i-*held* and *old*-l_i-*other*, and therefore precisely one of the four transitions is enabled in the `check` construction. Second, when a `pure` block is contained within a loop (for instance, `loop { block { pure { break } } }`), we must ensure `break` statements clean up the snapshots' tokens before exiting the `pure` block; this explains the rest of the construction for `break` in the diagram.

4.5 Defining pure-causal atomicity

As described above, four potential kinds of traces are possible: a trace may fire `purity` and so check the body for purity; a trace may fire `atomicity` and exit abruptly, and so check the program for causal atomicity; a trace may fire `skip` and model the normal termination of the body, and so check the program for causal atomicity; or a trace may fire `atomicity` and exit normally, in which case the results are unneeded (any real errors will be found either by purity checking or by abrupt termination; the remainder are false positives)[3].

To formalize pure-causal atomicity, we again want no event to come "between" two events in an `atomic` block, but now we need to reason solely about the proper kinds of traces. Therefore, define a *non-pure trace* as one where

$$\forall e \in \text{TRANS}(\texttt{pure}\ \ s) \,.\, \lambda(e) = \texttt{skip} \vee \\ \lambda(e) = \texttt{atomicity} \wedge \nexists e' \in \text{CURRENT}(\texttt{pure}\ \ s, e) \,.\, \lambda(e') = t_{checkPurity}$$

where CURRENT(`pure` s, e) is the set of events in one contiguous execution TRANS(`pure` s) that contains the event e. This definition selects the non-"vacuous" traces, exactly those which fire the `atomicity` or `skip` transitions and, when executing the body of a pure block, excecute a `break` statement.

[3] There is a subtlety involving infinite traces, which may run forever without terminating or breaking (for example, `pure` $\{ X := 42;$ `loop { skip } }`, which is not pure but will neither terminate nor break). To soundly reject such programs, we pragmatically require that *some* final transition always be reachable from every place in a pure block, which still permits all purity idioms we have so far encountered.

Only in these traces will our conclusions about pure-causal atomicity be valid. Formally, let P be a program, and let S be the subnet from the translation of some `atomic` block in P. Then S is *pure-causally atomic* if there does not exist a non-pure trace where

$$\exists e_1^T, e_2^{T'}, e_3^T \in \mathcal{E} \,.\, e_1 \in \mathrm{START}(S) \wedge e_1 \preceq e_2 \preceq e_3 \wedge \nexists e^T \in \mathrm{END}(S) \,.\, e_1 \preceq e \preceq e_3$$

Note that the definition of purity used by pure-causal atomicity is conservative: writing c to `x` when `x` already holds c changes no state but is considered impure (recall, our translation does not model the values of variables); additionally, `break`ing unnecessarily after a pure but atomic action may lead to atomicity violations. A more robust notion of purity would only change the translation slightly, and not change the definition of pure-causal atomicity.

4.6 Computing pure-causal atomicity: Coloring the net

To extend our analysis to support purity, tokens in our net actually have a color drawn from the set $\{ A, B, Y, R \} \times \{ n, o, b, m \}$. The first component tracks causal atomicity and is propagated using the rules described above. The second component tracks purity and its integration with pure-causal atomicity. Initially, all tokens are colored n (not in pure). On entry into a pure block, one of three transitions must be taken (all rules leave the first component unchanged):

- If `skip` fires at the start of the block, the purity color is left at n.
- If `atomicity` fires, the purity color is changed to m (must break), indicating this must be a non-pure trace. Since we are checking atomicity, we permit all the color changes described in Section 4.3. If there is a violation of causal atomicity, at least one variable or lock place must be colored R; we therefore add a transition from every variable or lock place to `ERROR`, coloring it R when the input place turns R. If we get to the `check-purity` transition, the trace succeeds vacuously, as it was not properly a non-pure trace (in the sense above), and no conclusions can be drawn. To handle this possible non-termination of the trace, we disable the transitions to `ERROR` until all `pure` blocks have terminated, at which time we are correct to report the violation.
- Finally, if `purity` fires at the start of the block, we turn the mark o (okay). This trace checks for purity, so we *disable* the atomicity coloring rules—the only way this trace can reach `ERROR` is if the block is not causally pure. The color o remains unchanged until a variable write, at which point it turns b (bad). When the block terminates normally, the checkpoint confirms that the locks have not been changed, and turns `ERROR` R otherwise. When `check-purity` is reached, a b token turns R and flows to `ERROR`, while an o token causes the trace to abort (purity checking succeeded on this trace, but all further atomicity checking is invalid since we ignored potential causal dependencies).

Similar to our informal justification for the causal-atomicity coloring rules above, the purity coloring rules for purity n and m pick out precisely the non-pure traces such that the atomicity colors again correspond to these events; additionally the coloring rules for o and b identify any purity violations.

5 Implementation

We built a prototype checker for the algorithm presented above, using CPN-Tools [12, 18]. Our toolchain accepts a program in a concrete syntax resembling Figure 6 and compiles it to a set of Petri nets, each designed to test a single `atomic` block of the source program. These nets are each read by CPN-Tools, which in turn constructs the state-space model. Each net contains the pure-causal atomicity reachability query, written in Standard ML using the library functions exported by the tool. If any of the nets satisfy the query, i.e., `ERROR` is colorable with a red token, then the source program is not pure-causally atomic. If all nets fail the query, the source program is pure-causally atomic.

Our implementation verifies all the examples in this paper as atomic (except the first, which is correctly rejected as not atomic), and correctly rejects variants of the examples that break atomicity. As in [1], we leave the extending of our approach to full-scale languages to future work, but we see no new fundamental problems. The toolchain and examples are available from our website [14, 15].

6 Expressiveness of pure-causal atomicity

The preceding sections have defined our pure-causal atomicity analysis, and explained how we compute the results for a given program. We showed in Section 2.4 that pure-causal atomicity can check examples no prior system could. The following two results show our translation to Petri nets also does not deem any programs non-atomic that the type-system approache validates. We state the formal results here, and sketch the essential ideas of the proofs. Properly formalizing these results requires a full definition of the type systems for purity and (pure-)reducible atomicity (see the technical report [14] for full details):

Theorem 1. *For every reducibly-atomic program P that has no* `pure` *blocks, all* `atomic` *blocks in P are causally atomic when translated into Petri nets.*

Sketch of Proof. By structural induction on each statement s in P; we strengthen the induction to show that if s type-checks as reducibly atomic (recall the type system ascribes a mover to every statement, not just `atomic` blocks) and all substatements of s are causally atomic, then s itself is causally atomic. We show the most interesting case here, of sequencing two statements.

By construction, every transition in TRANS(s) has a neighborhood consisting solely of places in s's thread, variable places (if it is a variable access) or lock places (if it is a lock operation). Therefore, causal dependencies between threads can only occur through actual contention over shared resources.

Consider the sequence $s_1; s_2$ appearing in the source program. Assume inductively that both statements type-check, that their individual translations are causally atomic, and that $s_1; s_2$ type-checks as atomic. We proceed by contradiction, assuming that $s_1; s_2$ is not causally atomic when translated into a Petri net. It follows that if the TRANS($s_1; s_2$) is to *not* be causally atomic, then there must be some events $e_1^T \in$ TRANS(s_1), $e_3^T \in$ TRANS(s_2), and some other event $e_2^{T'}$ such that $e_1 \preceq e_2 \preceq e_3$. Pick e_1 to be the *last* event in TRANS(s_1) to still

be causally before e_2, and pick e_3 to be the *earliest* event in TRANS(s_2) after e_2. (We know e_1 and e_3 cannot both be in the same substatement, or else that substatement would not be causally atomic, contradicting our earlier assumption.) By the above paragraph, if all three events are causally dependent and in different threads, then they must access locks or variables. Suppose all three events access the same lock (the variable case is similar). Then we know e_1 must be an `acquire`, and the lock must be held at least until e_2. If so, then e_2 cannot happen between the two events, because no other thread can manipulate a lock while it is held. We therefore have a contradiction: no such event e_2 can exist. □

Theorem 2. *For every pure-reducibly atomic program P, all* `pure` *blocks in P are actually pure and all* `atomic` *blocks in P are pure-causally atomic when translated into Petri nets.*

Sketch of Proof. The proof is very similar to that of the previous theorem, by structural induction on statements s in P; we strengthen the induction to show that if (1) s type-checks as pure-reducibly atomic, (2) all `pure` blocks in s are indeed pure (proven separately; see [14]), and (3) all substatements of s are pure-causally atomic, then s itself is pure-causally atomic. We sketch the key new case handling `pure` blocks.

Assuming that a pure block $s = \texttt{pure}\,\{\, s' \,\}$ is pure-reducibly atomic, suppose that TRANS(s) is not pure-causally atomic; we show this leads to a contradiction. Looking at the typing rules for pure-reducible atomicity, we are guaranteed by (1) that s' itself is pure-reducibly atomic; by (2) we know that s' is indeed pure and, using (3), by induction we therefore know TRANS(s') is pure-causally atomic, i.e., there does not exist a non-pure trace where events e_1 and e_3 in TRANS(s') satisfy the condition in pure-causal atomicity. Therefore, if there *is* a violation of the pure-causal atomicity of TRANS(s), at least one of e_1 and e_3 must correspond to transitions *not* in s'.

Examining the construction of TRANS(`pure` s'), we see that the only transitions not in TRANS(s') are part of the constructions we introduced to check purity. Crucially, each of these transitions is causally dependent *only* on other transitions in the current thread (this is why we needed the l_i-*held* and l_i-*other* places to be thread-local—the snapshot and checkpoint constructions can access them without any dependence on other threads); said another way, they are *independent* of all other threads. As a consequence, no event corresponding to these transitions can create a causal dependence with other threads, contradicting our assumption that TRANS(s) is not pure-causally atomic. □

7 Conclusion

We have defined a new notion of *pure-causal atomicity*, and shown that it extends both causal atomicity and pure-reducible atomicity in expressive power. It uses a non-trivial encoding of purity checking in Petri nets, and shows how to incorporate purity checking nearly orthogonally to atomicity checking. We formally show that causal atomicity is a conservative extension of reducible atomicity, and that pure-causal atomicity likewise extends pure-reducible atomicity.

References

1. Farzan, A., Madhusudan, P.: Causal atomicity. In: Computer Aided Verification. (2006) 315–328
2. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity. Software Engineering, IEEE Transactions on **31**(4) (2005) 275–291
3. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2003) 338–349
4. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, New York, NY, USA, ACM Press (2005) 47–58
5. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2005) 83–94
6. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2005) 61–71
7. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12) (1975) 717–721
8. Flanagan, C.: Verifying commit-atomicity using model-checking. In: SPIN. (2004) 252–266
9. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1-2) (2000) 167–188
10. Hatcliff, R.J., Dwyer, M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Verification, Model Checking, and Abstract Interpretation. (2004) 175–190
11. Lodaya, K., Mukund, M., Ramanujam, R., Thiagarajan, P.S.: Models and logics for true concurrency. Technical Report IMSc/90/12, The Institute of Mathematical Sciences, Madras, INDIA (1990)
12. Jensen, K., Kristensen, L., Wells, L.: Coloured petri nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer **9**(3, 4) (June 2007) 213–254
13. Flanagan, C., Qadeer, S.: Types for atomicity. In: TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation, New York, NY, USA, ACM Press (2003) 1–12
14. Lerner, B., Grossman, D.: Purifying causal atomicity. Technical report, University of Washington Computer Science and Engineering (2008)
15. Lerner, B.: `http://www.cs.washington.edu/homes/blerner/index.html`
16. Pugh, W.: `http://www.cs.umd.edu/~pugh/java/memoryModel/`
17. Jensen, K.: Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1. Springer-Verlag, London, UK (1996)
18. `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`