Table of Contents

Purifying Causal Atomicity 1
Benjamin S. Lerner and Dan Grossman
1 Intuition and introduction 1
2 Terminology & Notation 2
2.1 Language 2
2.2 Effect-system related definitions 2
2.3 Petri-net related definitions 4
3 Petri-fied Effects: Extending atomicity from syntax to dataflow 11
3.1 Refining the definition of causal atomicity 12
3.2 Enumerating the operations of movers 14
3.3 Sufficiency of our race detector 16
3.4 Causal atomicity of sequenced statements 20
3.5 Causal atomicity encompasses effect-based atomicity 23
4 Pure-Sml: distilling atomicity from impure sources 25
4.1 Language 25
4.2 Effect-system related definitions 25
4.3 Petri-net related definitions 26
5 Checking causal atomicity with purity 33
5.1 Defining pure-causal atomicity 33
5.2 Causal purity 36
5.3 Enumerating the operations of movers 39
5.4 Sufficiency of our race detector 39
5.5 Pure-causal atomicity of sequenced statements 45
5.6 Pure-causal atomicity encompasses effect-based abstract atomicity 48
6 Implementing the definition of pure-causal atomicity 50

Purifying Causal Atomicity

Benjamin S. Lerner and Dan Grossman

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

Abstract. Atomicity has been studied extensively as a tool for simplifying a programmer's understanding of concurrent code. The challenge of atomicity analysis is to precisely find those code sections of interest that do obey an atomicity discipline. In this paper, we present an extension to Farzan and Madhusudan's work on Causal Atomicity [1], adapting the purity analysis proposed by Flanagan et. al. [2] to the Petri-net setting. Our work is *compositional*—a different purity analysis could be implemented with minimal extra effort, and similarly another atomicity criterion could be checked without changing the purity translation—and *compatible*—the analysis of any program that does not use purity annotations is trivially equivalent to the original analysis.

1 Intuition and introduction

In [3], Flanagan and Qadeer introduce an *effect system* for tracking atomicity. Their system is based on Lipton's notion of *movers*, indicating in which directions primitive operations can commute with those of other threads. This analysis is sound, but is incomplete for several reasons. One of those weaknesses is addressed in [2], where they add a purity analysis to their effect system. Intuitively, pure blocks of code—those that do not modify the global state—leave behind no evidence that they ran, and hence should not affect the perceived atomicity of concurrent code. Their purity analysis performs effect masking over these blocks, making them "disappear" to the atomicity effect system.

An entirely different approach is taken by [1], in which Farzan and Madhusudan model a program by a *Petri net*, a graph-based model of computation in which control is modeled by a set of active *places*, and which flows through *transitions* that are enabled only when all their prerequisite places are active and that move control from their preconditions to their postconditions. Here, they encode programs by various widgets in the graph, and define atomicity as a reachability property of the graph. This model avoids the notion of movers entirely, thereby avoiding the need to worry about reducibility. But it still suffers from imprecision due to pure code.

In this paper, we adapt the purity analysis to the Petri-net formalism, showing how to augment the precision of the analysis in a compositional way. We show three things in this paper:

1. We show that the Petri-net model of causal atomicity is strictly more powerful than the effect-system model of reducible atomicity. That is, every pro-

Syntax of SML

$$
\begin{array}{rrcl}
P \in & Prog & ::= & \cdot \mid s || P \\
s \in & Stmt & ::= & x := e \mid s \,;\, s \mid \texttt{if}\ e\ s\ s \mid \texttt{while}\ e\ s \mid \texttt{atomic}\ s \mid \texttt{skip} \\
 & & \mid & \texttt{acquire}\ l \mid \texttt{release}\ l \\
e \in & Exp & ::= & c \mid x \mid p(\bar{e}) \\
x \in & Var & = & UnstableVar \uplus StableVar \\
c \in & Const & = & \mathbb{Z} \\
fn \in & Prim & = & ArithPrim \uplus LogicPrim \\
l \in & Lock & &
\end{array}
$$

Fig. 1: The language SML, with a few slight cosmetic changes to more closely match CAT and CAP

gram that effect-checks under the system in [3] can pass as causally atomic under the model in [1].

2. We encode the purity effect system of [2] as widgets in a Petri net, reducing the problem of affirming purity to a reachability problem over the net.
3. We show how to modify the atomicity analysis in [1] to include our purity analysis, running both analyses simultaneously over the same net and using the purity results to improve the atomicity ones.

2 Terminology & Notation

2.1 Language

Our example language is a hybrid of SML, the statement language used in [1] and CAT, the expression language used in [3]. As presented, CAT is more flexibly expressive than SML: it permits definitions of functions, function calls, and dynamic spawning of threads. To compare the two languages, we restrict all spawned threads to the top level, and remove function definitions and calls. We call this restricted space the set of *compatible* programs. Compatible programs are trivially translatable into the SML version presented here. The language is presented in Figure 1.

2.2 Effect-system related definitions

The race detection analysis used with CAT is not specified; it is supplied externally. For this translation, we must assume a race detection analysis that is computable by Petri nets, and in particular, this requires that the analysis be value-insensitive. We will assume a sound race analysis in which variables are marked as race-free when they are guaranteed to be protected by some fixed lock at all times; for the purposes of this paper we define a specific, simple and

$$\begin{aligned} \mathit{Atomicity} &= \{\bot, B, L, R, A, \top\} \\ \Gamma \ &: \ (\mathit{Prim} \rightarrow \mathit{Atomicity}) \uplus \\ &\quad (\mathit{Var} \rightarrow \{\bullet\} \uplus \mathit{Lock}) \\ \Sigma &\subseteq \mathit{Lock} \end{aligned}$$

$a;b$	$\bot$	B	L	R	A	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
B	$\bot$	B	L	R	A	$\top$
R	$\bot$	R	A	R	A	$\top$
L	$\bot$	L	L	$\top$	$\top$	$\top$
A	$\bot$	A	A	$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$

a	a^*
$\bot$	B
B	B
R	R
L	L
A	$\top$
$\top$	$\top$

Effect systems of Sml

$\boxed{\Gamma, \Sigma \vdash e : a}$

[Exp-Const]
$$\frac{}{\Gamma, \Sigma \vdash c : B}$$

[Exp-Var]
$$\frac{\Gamma(x) \in \Sigma}{\Gamma, \Sigma \vdash x : B}$$

[Exp-Var-Race]
$$\frac{\Gamma(x) = \bullet}{\Gamma, \Sigma \vdash x : A}$$

[Exp-Prim]
$$\frac{\Gamma, \Sigma \vdash e_i : a_i}{\Gamma, \Sigma \vdash fn(e_1, \ldots, e_n) : (a_1; \ldots; a_n; \Gamma(p))}$$

$\boxed{\Gamma, \Sigma \vdash s : a, \Sigma'}$

[Stmt-Skip]
$$\frac{}{\Gamma, \Sigma \vdash \texttt{skip} : B, \Sigma}$$

[Stmt-Seq]
$$\frac{\Gamma, \Sigma \vdash s_1 : a_1, \Sigma' \qquad \Gamma, \Sigma' \vdash s_2 : a_2, \Sigma''}{\Gamma, \Sigma \vdash s_1; s_2 : (a_1; a_2), \Sigma''}$$

[Stmt-Assign]
$$\frac{\Gamma, \Sigma \vdash e : a \qquad \Gamma(x) \in \Sigma}{\Gamma, \Sigma \vdash x := e : (a; B), \Sigma}$$

[Stmt-Assign-Race]
$$\frac{\Gamma, \Sigma \vdash e : a \qquad \Gamma(x) = \bullet}{\Gamma, \Sigma \vdash x := e : (a; A), \Sigma}$$

[Stmt-Atomic]
$$\frac{\Gamma, \Sigma \vdash e : a, \Sigma' \qquad a \sqsubseteq A}{\Gamma, \Sigma \vdash \texttt{atomic}\ \ s : a, \Sigma'}$$

[Stmt-While]
$$\frac{\Gamma, \Sigma \vdash e : a_e \qquad \Gamma, \Sigma \vdash s : a_s, \Sigma}{\Gamma, \Sigma \vdash \texttt{while}\ e\ s : (a_e; (a_s; a_e)^*), \Sigma}$$

[Stmt-If]
$$\frac{\Gamma, \Sigma \vdash e : a_e \qquad \Gamma, \Sigma \vdash s_1 : a_1, \Sigma \qquad \Gamma, \Sigma \vdash s_2 : a_2, \Sigma}{\Gamma, \Sigma \vdash \texttt{if}\ e\ s_1\ \ s_2 : (a_e; (a_1 \sqcup a_2)), \Sigma}$$

[Stmt-Acquire]
$$\frac{l \notin \Sigma}{\Gamma, \Sigma \vdash \texttt{acquire}\ l : R, \Sigma \cup \{\, l \,\}}$$

[Stmt-Release]
$$\frac{l \in \Sigma}{\Gamma, \Sigma \vdash \texttt{release}\ l : L, \Sigma \setminus \{\, l \,\}}$$

$\boxed{\vdash P : ok}$

[Prog-OK]
$$\frac{\forall 1 \leq i \leq n. \Gamma, \emptyset \vdash s_i : a_i, \Sigma_i}{\vdash s_1 || \cdots || s_n : ok}$$

Fig. 2: The corresponding effect systems for atomicity and race detection

conservative race detector. More generally, we need a specific property we call "sufficiency" that we will define later.

The atomicity analysis defined for Cat uses an effect system to model statements as both-, left- or right-movers, as atomic statements, or as non-atomic statements. (For now, ignore the $\bot$ atomicity, and treat B as the bottom element of the lattice. We will need $\bot$ when extending the system with purity analyses.) The corresponding analysis for Sml, along with the race detector it uses, is defined in Figure 2.

2.3 Petri-net related definitions

We follow the notation in [1] fairly closely, specializing some of their general definitions to our specific usage.

Definition 1. *Petri nets, places, transitions, flow relation.*

A *Petri net* is a triple $N = (P, T, F)$, where P is a set of *places*, T (disjoint from P) is a set of *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*.

The union of two Petri nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ is defined componentwise, as $N_1 \cup N_2 \stackrel{\text{def}}{=} (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$.

In our usage, places will mark program points, variables, locks, and various administrative details, transitions will correspond to program statements or expressions, and the flow relation connects program points (and variables etc.) by way of program statement transitions.

Definition 2. *Preconditions, postconditions, neighborhood, independence relation, dependence relation.*

The *preconditions* of a transition t are all places which flow to t; similarly the *postconditions* are all places to which t flows; they are defined as

$$
\begin{aligned}
{}^{\bullet}t &\stackrel{\text{def}}{=} \{p \in P \mid (p, t) \in F\} \\
t^{\bullet} &\stackrel{\text{def}}{=} \{p \in P \mid (t, p) \in F\}
\end{aligned}
$$

The *neighborhood* of a transition t is the union of its pre- and post-conditions.

Two transitions are *independent* if and only if their neighborhoods do not overlap, and we define the *independence relation* accordingly:

$$
I \stackrel{\text{def}}{=} \{(t, t') \in T \times T \mid ({}^{\bullet}t \cup t^{\bullet}) \cap ({}^{\bullet}t' \cup t'^{\bullet}) = \emptyset\}
$$

Two transitions are *dependent* if they are not independent — that is, the neighborhoods of two dependent transitions overlap. The dependence relation is defined as

$$
D \stackrel{\text{def}}{=} (T \times T) \setminus I
$$

Definition 3. *Marking of a net, enabled transitions, transition relation, firing sequence, 1-safety.*

A *marking* is a subset of the places in the net; each place in the marking is marked with a *token*. More precisely, this is a *1-bounded*, *colorless* marking. The general definition permits marks to have colors (so a marking would be a partial function from P to $Colors$), and permits multiple tokens on a given place simultaneously (so a marking would be a partial function from P to multi-subsets of $Colors$).

A transition t is *enabled* under a marking M if ${}^{\bullet}t \subset M$, that is, all preconditions of the transition must be marked. If so, the transition can fire (though it is not required to do so), and we say $M \xrightarrow{t} M'$ where $M' = (M \setminus {}^{\bullet}t) \cup t^{\bullet}$. This removes the marks from the preconditions and marks the postconditions; no other marks are changed. Let $\xrightarrow{*}$ be the reflexive, transitive closure of $\rightarrow$. (Note that this definition of $\xrightarrow{t}$ presupposes that the marking is 1-bounded, since it uses set operations instead of multi-set operations; the general definition would have to handle colors and multiplicities in ${}^{\bullet}t$ and M.)

A *firing sequence* is a sequence of transitions $t_1 t_2 \ldots$ provided we have a sequence of markings $M_0 M_1 \ldots$ where M_0 is the initial marking on the net, and $M_i \xrightarrow{t_i} M_{i+1}$.

A Petri net is *1-safe* if every firing sequence leaves the marking 1-bounded, that is, there does not exist a firing sequence for which at some marking M_i, a transition t_i is enabled that would leave more than 1 token on some place. 1-safety is crucial to our translations later involving purity; all our translations from programs to Petri nets will preserve this property.

Definition 4. *Trace, events, causal order, labeling.*

A *trace* of a Petri net is T-labeled poset $(\mathcal{E}, \preceq, \lambda)$ with certain restrictions. $\mathcal{E}$ is a finite or a countable set of *events*, the statements that occur when executing a program. Every event is unique, but multiple events can correspond to the same transition in the Petri net; we use the labeling function $\lambda : \mathcal{E} \rightarrow T$ to give names to the events. (In this way we can distinguish, for instance, the event of entering a loop from the event of re-entering that loop for the fourth time.) Finally, $\preceq \subseteq \mathcal{E} \times \mathcal{E}$ is a partial order on $\mathcal{E}$, called the *causal order*. Given $\preceq$, define the relation $\prec$ as $e_1 \prec e_2 \stackrel{\text{def}}{=} e_1 \preceq e_2 \wedge e_1 \neq e_2$. Define the *immediate causal order* relation by

$$e_1 \prec\!\!\cdot\, e_2 \stackrel{\text{def}}{=} e_1 \prec e_2 \wedge \nexists f \in \mathcal{E}.(e_1 \prec f \prec e_2)$$

The following conditions relate the causal order to the labeling function and the dependence relation:

- $\forall e \in \mathcal{E}, \{e' \in \mathcal{E} \mid e' \preceq e\}$ is finite. In other words, there are only finitely many events causally before e.
- $\forall e, e' \in \mathcal{E}. e \prec\!\!\cdot\, e' \Rightarrow \lambda(e) D \lambda(e')$. If two events are immediately causally ordered, the corresponding transitions must be dependent.
- $\forall e, e' \in \mathcal{E}. \lambda(e) D \lambda(e') \Rightarrow (e \preceq e' \vee e' \preceq e)$ If two events have dependent labels, they must be causally ordered.

(In general, traces are more generally defined, and need not use the exact choices here that labels name transitions and the depedence relation identifies "adjacent" transitions. For this work, this restricted definition will suffice.)

In the rest of this paper, we will only consider traces that correspond to firing sequences. In other words, given a trace $Tr = (\mathcal{E}, \preceq, \lambda)$, there is some total ordering of the events $e_1, e_2, \ldots$ that respects the partial order, such that the sequence $M_0 \xrightarrow{\lambda(e_1)} M_1 \xrightarrow{\lambda(e_2)} M_2 \cdots$ is a valid firing sequence of the net, given an initial marking M_0. (This is not the precise technical definition of "traces corresponding to firing sequences", but suffices to give intuition. We won't be using the lowest-level definitions of traces in this paper.)

The definition of the immediate causal relation implies the following corollaries:

Corollary 1.

$$e_1 \prec e_3 \implies \exists e_2 . e_1 \prec\!\!\cdot\; e_2 \preceq e_3$$
$$e_1 \prec e_3 \implies \exists e_1 . e_1 \preceq e_2 \prec\!\!\cdot\; e_3$$

Proof. We prove the first statement; the second is symmetric. Consider the set of events $S = \{ e \mid e_1 \prec e \preceq e_3 \}$. The set S must not be empty since $e_3 \in S$; it must also be finite since we are drawing from a finite pool of events preceding e_3 by the first condition on traces, above. Define the set $T = \{ e \in S \mid \nexists e' \in S . e' \preceq e \}$. The set T must not be empty because $\preceq$ is a partial order, and hence must have least elements: if every element in a finite set had a predecessor, then by the pigeonhole principle some element would have itself as an ancestor, violating the partial ordering. Therefore $\exists e_2 \in T$ satisfying the above condition. □

Corollary 2.

$$e_1 \prec e_3 \wedge e_1 \prec\!\!\cdot\; e_2 \wedge (\nexists e_2' \neq e_2 . e_1 \prec\!\!\cdot\; e_2' \preceq e_3) \implies e_1 \prec\!\!\cdot\; e_2 \preceq e_3$$
$$e_1 \prec e_3 \wedge e_2 \prec\!\!\cdot\; e_3 \wedge (\nexists e_2' \neq e_2 . e_1 \preceq e_2' \prec\!\!\cdot\; e_3) \implies e_1 \preceq e_2 \prec\!\!\cdot\; e_3$$

Proof. Again we prove the first statement; the second is symmetric. By the above corollary and the first premise, $\exists f . e_1 \prec\!\!\cdot\; f \preceq e_3$. If $e_2 = e_3$, then we are trivially done, so assume $e_2 \neq e_3$. If $f = e_2$, we are trivially done, so assume $f \neq e_2$. But this contradicts our third premise. □

Definition 5. *Translation of a program.*

Given a program P in our language, denote by TRANS(P) the Petri net constructed in Figure 4 corresponding to that program. (A graphical version of this translation is drawn in Figure 3.) For a given statement $s \in P$ (note: this s is not necessarily a top-level statement as denoted in PROG-OK, but rather any statement anywhere in P), denote also by TRANS(s) the subnet of the Petri net corresponding to that statement, and similarly for a given expression $e \in P$. Finally, in a slight abuse of notation, we will say that places or transitions are "in" a given net: for a net $N = (P, T, F)$, $p \in N \Leftrightarrow p \in P$, and similarly for transitions t. To keep the notation concise, we will say that an event $e \in_\lambda N \Leftrightarrow \lambda(e) \in N$.

As we'll show later, we can use a slightly different translation than this one (which was taken from [1]). The differences appear in just two statement forms. First, the t_{begin} and t_{end} transitions when translating `atomic` blocks are not needed to formulate a definition of causal atomicity (see below). Second, to achieve this revised definition, we need to add an explicit loop-head transition when translating `while` loops; it is marked t_{head}. These changes are denoted by dashed lines in the diagram. The revised pseudocode implementing these changes is shown in Figure 5.

Definition 6. *First, Last, Start, End sets of events.*

In the proofs that follow, it will be convenient to talk about the first or last transitions that can possibly fire within a subnet. To be precise, define

$$\begin{aligned}
\mathrm{START}(s) &\stackrel{\mathrm{def}}{=} \{t \in \mathrm{TRANS}(s) \mid \nexists t' \in \mathrm{TRANS}(s). t'^{\bullet} \cap {}^{\bullet}t \neq \emptyset\} \\
\mathrm{END}(s) &\stackrel{\mathrm{def}}{=} \{t \in \mathrm{TRANS}(s) \mid \nexists t' \in \mathrm{TRANS}(s). t^{\bullet} \cap {}^{\bullet}t' \neq \emptyset\}
\end{aligned}$$

Examining figure 4 shows inductively that for any translated program statement or expression, the START and END sets are non-empty.

We will also want to talk about the first events causally after a given event, or the last events causally before a given event. For a given trace $tr = (\mathcal{E}, \preceq, \lambda)$, and some event $h \in \mathcal{E}$, define

$$\begin{aligned}
\mathrm{LAST}(s, h, tr) &\stackrel{\mathrm{def}}{=} \{e \in_\lambda \mathrm{TRANS}(s) \mid (e \preceq h) \wedge \\
&\qquad\qquad \nexists f \in_\lambda \mathrm{TRANS}(s).(e \prec f \preceq h)\} \\
\mathrm{FIRST}(s, h, tr) &\stackrel{\mathrm{def}}{=} \{e \in_\lambda \mathrm{TRANS}(s) \mid (h \preceq e) \wedge \\
&\qquad\qquad \nexists f \in_\lambda \mathrm{TRANS}(s).(h \preceq f \prec e)\}
\end{aligned}$$

When the trace is clear from the context, the argument tr will be left implicit. Typically, the event h will be such that $h \notin_\lambda \mathrm{TRANS}(s)$, and in fact will not be in the (translation of the) same thread as s.

Finally, we want to talk about the current execution of a given statement. For a given trace tr, statement s, and event $h \in_\lambda \mathrm{TRANS}(s)$, define

$$\begin{aligned}
\mathrm{CURRENT}(s, h) \stackrel{\mathrm{def}}{=} \{e \in_\lambda \mathrm{TRANS}(s) \mid\ & \exists e_b \in_\lambda \mathrm{START}(s). \\
& (e_b \preceq e \wedge \nexists e_e \in_\lambda \mathrm{END}(s). e_b \preceq e_e \preceq h) \wedge \\
& \exists e_e \in_\lambda \mathrm{END}(s). \\
& (e \preceq e_e \wedge \nexists e_b \in_\lambda \mathrm{START}(s). h \preceq e_b \preceq e_e)\}
\end{aligned}$$

In other words, the current execution of a given statement is comprised of all events after the most recent starting events before h, and before the soonest ending events after h.

Definition 7. *Causal atomicity [1].*

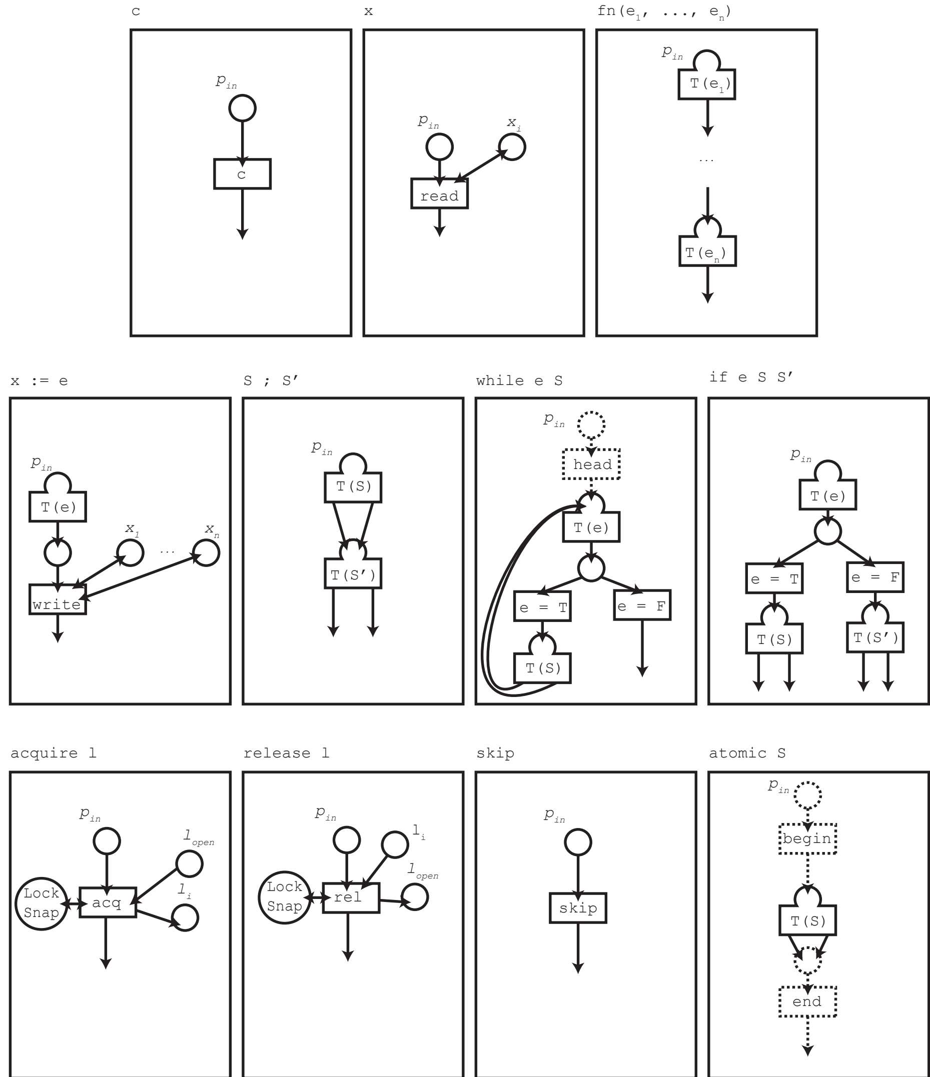

Fig. 3: Translation of programs from CAT into Petri nets. (To save space, the function TRANS is denoted by `T` in the diagram.) Dashed circles, boxes and arrows indicate components that differ from the translation presented in [1].

$$\begin{aligned}
&\text{TRANS}_e : (Exp, V, tid) \to ((P,T,F), p_{in} \in P, out \subset T)\\
&\quad \text{TRANS}_e(c, V, tid) = ((P,T,F), p_{in}, \{t_c\}) \text{ where}\\
&\qquad P = \{p_{in} \text{ fresh}(tid)\}\\
&\qquad T = \{t_c \text{ fresh}(tid)\}\\
&\qquad F = \{(p_{in}, t_c)\}\\
&\quad \text{TRANS}_e(x, V, tid) = ((P,T,F), p_{in}, \{t_x\}) \text{ where}\\
&\qquad P = \{p_{in} \text{ fresh}(tid), x_{tid} \in V\}\\
&\qquad T = \{t_x \text{ fresh}(tid)\}\\
&\qquad F = \{(p_{in}, t_x), (x_{tid}, t_x), (t_x, x_{tid})\}\\
&\quad \text{TRANS}_e(fn(e_1, \dots, e_n), V, tid) = ((P,T,F), p_1, out_n\}) \text{ where}\\
&\qquad ((P_i, T_i, F_i), p_i, out_i) = \text{TRANS}_e(e_i, V, tid) \quad \text{for } 1 \le i \le n\\
&\qquad P = \textstyle\bigcup_{1 \le i \le n} P_i\\
&\qquad T = \textstyle\bigcup_{1 \le i \le n} T_i\\
&\qquad F = \textstyle\bigcup_{1 \le i \le n} F_i \cup \bigcup_{1 \le i < n} \{(o_i, p_{i+1}) \mid o_i \in out_i\}
\end{aligned}$$

$$\begin{aligned}
&\text{TRANS}_s : (Stmt, V, L, tid) \to ((P,T,F), p_{in} \in P, out \subset T)\\
&\quad \text{TRANS}_s(\texttt{skip}, V, L, tid) = ((P,T,F), p_{in}, \{t_{skip}\}) \text{ where}\\
&\qquad P = \{p_{in} \text{ fresh}(tid)\}\\
&\qquad T = \{t_{skip} \text{ fresh}(tid)\}\\
&\qquad F = \{(p_{in}, t_{skip})\}\\
&\quad \text{TRANS}_s((s_1; s_2), V, L, tid) = ((P,T,F), p_1, out_2) \text{ where}\\
&\qquad ((P_i, T_i, F_i), p_i, out_i) = \text{TRANS}_s(e_i, V, L, tid) \quad \text{for } i = 1, 2\\
&\qquad P = P_1 \cup P_2\\
&\qquad T = T_1 \cup T_2\\
&\qquad F = F_1 \cup F_2 \cup \{(o_1, p_2)\} \mid o_1 \in out_1\}\\
&\quad \text{TRANS}_s(x := e, V, L, tid) = ((P,T,F), p_{in}, \{t_x\}) \text{ where}\\
&\qquad ((P_e, T_e, F_e), p_{in}, out_e) = \text{TRANS}_e(e, V, tid)\\
&\qquad P = P_e \cup \{p_e \text{ fresh}(tid)\} \cup \{x_i \mid x_i \in V\}\\
&\qquad T = T_e \cup \{t_x \text{ fresh}(tid)\}\\
&\qquad F = F_e \cup \{(o_e, p_e) \mid o_e \in out_e\} \cup \{(p_e, t_x)\} \cup\\
&\qquad\qquad \{(x_i, t_x) \mid x_i \in V\} \cup \{(t_x, x_i) \mid x_i \in V\}\\
&\quad \text{TRANS}_s(\texttt{atomic } s, V, L, tid) = ((P,T,F), p_{in}, \{t_{end}\}) \text{ where}\\
&\qquad ((P_s, T_s, F_s), p_s, out_s) = \text{TRANS}_s(s, V, L, tid)\\
&\qquad P = P_s \cup \{p_{in} \text{ fresh}(tid), p_{end} \text{ fresh}(tid)\}\\
&\qquad T = T_s \cup \{t_{begin} \text{ fresh}(tid), t_{end} \text{ fresh}(tid)\}\\
&\qquad F = F_s \cup \{(p_{in}, t_{begin}), (t_{begin}, p_s), (p_{end}, t_{end})\} \cup \{(o_s, p_{end}) \mid o_s \in out_s\}
\end{aligned}$$

Fig. 4: Pseudocode of TRANS(*Prog*), TRANS(*Exp*) and TRANS(*Stmt*). All "fresh(*tid*)" places and transitions are meant to be new and marked as part of thread *tid*, even if their names have appeared before in P or T.

$$\begin{aligned}
&\text{TRANS}_s(\texttt{if}\ e\ s_1\ s_2, V, L, tid) = ((P,T,F), p_{in}, out_1 \cup out_2) \text{ where}\\
&\qquad ((P_e,T_e,F_e), p_{in}, out_e) = \text{TRANS}_e(e, V, tid)\\
&\qquad ((P_i,T_i,F_i), p_i, out_i) = \text{TRANS}_s(s_i, V, L, tid) \quad \text{for } i = 1,2\\
&\qquad P = P_e \cup P_1 \cup P_2 \cup \{p_e \text{ fresh}(tid)\}\\
&\qquad T = T_e \cup T_1 \cup T_2 \cup \{t_t \text{ fresh}(tid), t_f \text{ fresh}(tid)\}\\
&\qquad F = F_e \cup F_1 \cup F_2 \cup \{(o_e, p_e) \mid o_e \in out_e\} \cup\\
&\qquad\qquad \{(p_e, t_t), (p_e, t_f), (t_t, p_1), (t_f, p_2)\}\\
&\text{TRANS}_s(\texttt{while}\ e\ s, V, L, tid) = ((P,T,F), p_{in}, \{t_f\}) \text{ where}\\
&\qquad ((P_e,T_e,F_e), p_{in}, out_e) = \text{TRANS}_e(e, V, L, tid)\\
&\qquad ((P_s,T_s,F_s), p_s, out_s) = \text{TRANS}_s(s, V, L, tid)\\
&\qquad P = P_e \cup P_s \cup \{p_e \text{ fresh}(tid)\}\\
&\qquad T = T_e \cup T_a \cup \{t_t \text{ fresh}(tid), t_f \text{ fresh}(tid)\}\\
&\qquad F = F_e \cup F_s \cup \{(o_e, p_e) \mid o_e \in out_e\} \cup \{(o_s, p_{in}) \mid o_s \in out_s\} \cup\\
&\qquad\qquad \{(p_e, t_t), (p_e, t_f), (t_t, p_s)\}\\
&\text{TRANS}_s(\texttt{acquire}\ l, V, L, tid) = ((P,T,F), p_{in}, \{t_{acq\ l}\}) \text{ where}\\
&\qquad P = \{p_{in} \text{ fresh}(tid), l_{open} \in L, l_{tid} \in L\}\\
&\qquad T = \{t_{acq\ l} \text{ fresh}(tid)\}\\
&\qquad F = \{(p_{in}, t_{acq\ l}), (l_{open}, t_{acq\ l}), (t_{acq\ l}, l_{tid})\}\\
&\text{TRANS}_s(\texttt{release}\ l, V, L, tid) = ((P,T,F), p_{in}, \{t_{rel\ l}\}) \text{ where}\\
&\qquad P = \{p_{in} \text{ fresh}(tid), l_{open} \in L, l_{tid} \in L\}\\
&\qquad T = \{t_{rel\ l} \text{ fresh}(tid)\}\\
&\qquad F = \{(p_{in}, t_{rel\ l}), (l_{tid}, t_{rel\ l}), (t_{rel\ l}, l_{open})\}
\end{aligned}$$

$\text{TRANS}_p : Prog \rightarrow ((P,T,F), M \subset P)$

$$\begin{aligned}
&\text{TRANS}_s(s_1 || \cdots || s_n) = ((P,T,F), M) \text{ where}\\
&\qquad V = \{x_i \mid x \text{ appears in some } s_j \wedge 1 \leq i \leq n\}\\
&\qquad L = \{l_{open} \mid l \text{ appears in some } s_j\} \cup \{l_i \mid l \text{ appears in some } s_j \wedge 1 \leq i \leq n\}\\
&\qquad ((P_i,T_i,F_i), p_i, out_i) = \text{TRANS}_s(s_i, V, L, i)\\
&\qquad P = \textstyle\bigcup_{1 \leq i \leq n} P_i\\
&\qquad T = \textstyle\bigcup_{1 \leq i \leq n} T_i\\
&\qquad F = \textstyle\bigcup_{1 \leq i \leq n} F_i\\
&\qquad M = \{l_{open} \mid l \text{ appears in some } s_j\} \cup \{p_i \mid 1 \leq i \leq n\}
\end{aligned}$$

Fig. 4: (cont.) Pseudocode of TRANS(*Prog*), TRANS(*Exp*) and TRANS(*Stmt*). All "fresh(*tid*)" places and transitions are meant to be new and marked as part of thread *tid*, even if their names have appeared before in P or T.

To denote that a transition t is in some thread tid, write t^{tid}. Similarly, to denote that an event e corresponds to a transition in thread tid, write $e^{tid} \Leftrightarrow \lambda(e) = t^{tid}$. Once stated, the superscripts will be elided unless needed for clarity.

A code block $B = \texttt{atomic } S$ in program P is *causally atomic* if and only if TRANS(P) does not have a trace tr for which the following condition holds:

$$\begin{array}{ll} \exists e^{T}_{begin}, e^{T}_{2}, f^{T'} \in \mathcal{E}.e_{begin} \preceq f \preceq e_2 & \text{where} \\ \lambda(e^{T}_{begin}) = t^{T}_{begin} & \text{such that} \\ T \neq T' \text{ and } \nexists e^{T}_{end} \in \mathcal{E}.(\lambda(e^{T}_{end}) = t^{T}_{end} \wedge e_{begin} \preceq e_{end} \preceq e_2) & \end{array}$$

The transitions t^{T}_{begin} and t^{T}_{end} are the two fresh transitions constructed in TRANS(`atomic` S) (as opposed to any similarly-named transitions constructed recursively for the body S of the atomic statement); T is the associated thread. This condition states that a block of code is atomic if there is an interleaving of events where f, an event from a second thread T', is causally after e_{begin}, the beginning of the atomic block in thread T, another event e_2 in thread T is causally after f, and the atomic block still has not finished.

Definition 8. *Sufficient race detector*

Suppose a sound race analysis is run over a program, and two accesses to the same variable x (from different threads) are shown to be race-free, and at least one is a write-access. When translated to Petri nets, there are two transitions t^{T}_{x} and $t^{T'}_{x}$ corresponding to the variable accesses. Then for any trace where both transitions fire, we have two events u^{T} and $v^{T'}$ corresponding to t^{T}_{x} and $t^{T'}_{x}$.

We require that two additional events exist, u^{T}_{sync} and $v^{T'}_{sync}$, such that either $u \preceq u_{sync} \preceq v_{sync} \preceq v$ or $v \preceq v_{sync} \preceq u_{sync} \preceq u$. The events u_{sync} and v_{sync} are synchronization events that ensure the race-freedom of the accesses. Any race analysis that ensures this requirement is called *sufficient*.

3 Petri-fied Effects: Extending atomicity from syntax to dataflow

We proceed in stages:

1. We define a slightly different notion of causal atomicity, and prove it equivalent to the original. This lets us modify the translation of programs into Petri nets to remove some extraneous transitions.
2. We show that if an expression or statement has an atomicity strictly stronger than A, there are restrictions on what operations it can perform.
3. We show that the race detector we defined is in fact sufficient for our needs.
4. We use the previous two results to show that if two statements are sequenced in the program, and the sequence effect-checks as atomic, and both subnets corresponding to the translations of the two statements are causally atomic, then the combination of their nets is causally atomic as well.
5. We use this to show that for any program that effect checks, all atomic blocks will indeed be causally atomic.

3.1 Refining the definition of causal atomicity

Lemma 1. *The following condition is equivalent to the definition of causal atomicity presented in [1]:*

A code block $B =$ `atomic` S *in program* P *is* causally atomic *if and only if* TRANS(P) *does not have a trace tr for which the following condition holds:*

$$\exists e_1^T \in_\lambda \text{START}(S), e_2^T, f^{T'} \in \mathcal{E}.e_1 \preceq f \preceq e_2 \qquad \textit{such that}$$
$$T \neq T' \textit{ and } \nexists e^T \in_\lambda \text{END}(S).e_1 \preceq e \preceq e_2$$

In other words, the explicit transitions t_{begin}^T *and* t_{end}^T *from the original definition are not needed.*

Proof. First, a technical side point: In order for this lemma to actually be true, we must ensure that the START set of any translated statement is never empty. In the original translation of `while` loops, this is not the case. Accordingly, we introduced the extra t_{head} transition into the translation of `while` loops, but we must now ensure that doing so does not change their causal atomicity, or else the translation is broken. Fortunately this is easy to confirm: since this transition immediately precedes only the translation of e, any trace that witnesses the non-atomicity of the loop-with-`head` translation would witness the non-atomicity of the loop-without-`head` translation, and vice versa. We spell this argument out in more detail in connection with the t_{begin} and t_{end} transitions for `atomic` blocks.

Denote by e_{begin}^T and e_{end}^T the events corresponding to the begin and end transitions, if they occur. First, we know that

$$\forall e \in \text{CURRENT}(B, e_{begin}).e \in_\lambda \text{START}(S) \Leftrightarrow e_{begin} \prec e)$$
$$\forall e \in \text{CURRENT}(B, e_{end}).e \in_\lambda \text{END}(S) \Leftrightarrow e \prec e_{end})$$

We prove both directions of this, focusing on the first statement (the second is symmetric):

($\Leftarrow$) If $e_{begin} \prec e$, then $\lambda(e_{begin}) D \lambda(e)$. But by construction, t_{begin} only connects to transitions in START(S) (and to transitions before TRANS(S), but any events occurring there are not in CURRENT(B, e_{begin})). Therefore $e \in_\lambda$ START(S).
($\Rightarrow$) If $e_1 \in_\lambda$ START(S) and $e_1 \in$ CURRENT(B, e_{begin}), then $e_{begin} \preceq e_1$, and by our corollary above, $\exists f \in \text{TRANS}(S).e_{begin} \prec f \preceq e_1$. If $f \neq e_1$ then this contradicts $e_1 \in$ START(S), since events in START(S) have no predecessors in TRANS(S).

Also note that

$$\forall e \in \text{CURRENT}(B, e_{begin}), \text{CURRENT}(B, e) = \text{CURRENT}(B, e_{begin})$$

We prove now both directions of the lemma:

($\Leftarrow$) Assume `atomic` S is not new-causally-atomic. Show that `atomic` S is not old-causally-atomic. Therefore, assume

$$\begin{aligned}
&\exists e_1, f, e_2 \in \mathcal{E}.e_1 \preceq f \preceq e_2 && \text{where} \\
&e_1 \in_\lambda \text{START}(S), \lambda(e_2) = t_2, \lambda(f) = t_3^{T'} && \text{such that} \\
&T \neq T' \text{ and } \nexists e^T \in_\lambda \text{END}(S).e_1 \preceq e \preceq e_2
\end{aligned}$$

By construction, we know that if $e_1 \in_\lambda \text{START}(S)$, then we must have $\exists e_{begin} \in \text{CURRENT}(B, e_1)$, since t_{begin} is connected to p_{in} of $\text{TRANS}(S)$. Therefore, $\text{CURRENT}(B, e_1) = \text{CURRENT}(B, e_{begin})$. We know that $\forall e \in \text{CURRENT}(B, e_{begin}).e \in_\lambda \text{START}(S) \Leftrightarrow e_{begin} \prec e$; in particular, $e_{begin} \prec e_1^T$. Therefore $e_{begin} \prec e_1 \preceq f \preceq e_2$, and so $e_{begin} \preceq f \preceq e_2$, as would be required by the old definition.

Suppose $\exists e_{end} \in \text{CURRENT}(B, e_{begin}).\lambda(e_{end}) = t_{end}$. We know that $\forall e \in \text{CURRENT}(B, e_{end}).e \in_\lambda \text{END}(S) \Leftrightarrow e \prec e_{end}$. Therefore if $\exists e_{end} \in_\lambda \mathcal{E}.e_1 \preceq e_{end} \preceq e_2$, then $e_1 \preceq e \preceq e_2$ for any $e \in_\lambda END(S)$ (using the first corollary above on e_1, e and e_{end}); the contrapositive gives us that $(\neg \forall e \in_\lambda \text{END}(S).e_1 \preceq e \preceq e2) \implies (\nexists e_{end} \in_\lambda \mathcal{E}.e_1 \preceq e_{end} \preceq e_2)$. From our assumption we have $\nexists e \in_\lambda \text{END}(S).e_1 \preceq e \preceq e_2$, which together with the above implication gives us the condition required by the old definition.

Therefore if an event f exists that satisfies this definition, then it also satisfies the original definition. Equivalently, if a block B is not new-causally-atomic, then it is not old-causally-atomic either.

($\Rightarrow$) Assume `atomic` S is not old-causally-atomic. Show that `atomic` S is not new-causally-atomic. Therefore, assume

$$\begin{aligned}
&\exists e_{begin}, f, e_2 \in \mathcal{E}.e_{begin} \preceq f \preceq e_2 && \text{where} \\
&\lambda(e_{begin}) = t_{begin}^T, \lambda(e_2) = t_2^T, \lambda(f) = t_3^{T'} && \text{such that} \\
&T \neq T' \text{ and } \nexists e_{end} \in \mathcal{E}.(\lambda(e_{end}) = t_{end}^T \wedge e_{begin} \preceq e_{end} \preceq e_2)
\end{aligned}$$

We know that $\forall e \in \text{CURRENT}(B, e_{begin}).e \in_\lambda \text{START}(S) \Leftrightarrow e_{begin} \prec e$, and we know that $\text{START}(S)$ is not empty. Therefore we must have that $\exists e_1^T \in_\lambda \text{START}(S).e_1 \preceq f \preceq e_2$ (using the second corollary above), as required by the new definition.

We know that $e_{begin} \preceq e_2$; additionally, we know from our assumption that $\nexists e_{end}^T.e^{begin} \preceq e_{end} \preceq e_2$. Therefore we have that $e_2 \in \text{CURRENT}(B, e_1)$. Suppose $e_2 \notin \text{CURRENT}(S, e_1)$. Then $e_2 = e_{begin}$ or $e_2 = e_{end}$, since there are no other events in $\text{CURRENT}(B, e_1) \setminus \text{CURRENT}(S, e_1)$. But clearly the first option is impossible, since we have $e_{begin} \preceq f \preceq e_2$, and those inequalities are actually strict. So $e_2 = e_{end}$. But this too yields a contradiction: it is easy to satisfy $\exists e_{end}^T.e_{begin} \preceq e_{end} \preceq e_2$ when $e_{end} = e_2$. So $e_2 \in \text{CURRENT}(S, e_1)$. Define $E = \{e \in \text{CURRENT}(S, e_1) \mid e \in_\lambda \text{END}(S)\}$. Then clearly, we must have $\forall e^T \in \text{CURRENT}(S, e_1).\nexists e' \in E.e' \preceq e$, since e' is one of the end events of $\text{CURRENT}(S, e_1)$. Therefore $\nexists e^T \in_\lambda \text{END}(S).e_1 \preceq e \preceq e_2$ as required.

Therefore if an event f exists that satisfies the original definition, then it also satisfies this definition. Equivalently, if a block B is not old-causally-atomic, then it is not new-causally-atomic either. □

From now on, we use the second definition of causal atomicity exclusively, and update our translation function TRANS to leave out the explicit transitions t_{begin} and t_{end} when translating `atomic` S, and to include the t_{head} transition when translating `while` loops.

The revisions to the translation function is shown in Figure 5. All other cases are unchanged.

$$
\begin{aligned}
&\mathrm{TRANS}_s(\texttt{while}\ e\ s, V, L, tid) = ((P,T,F), p_{in}, \{t_f\}) \text{ where}\\
&\qquad ((P_e,T_e,F_e), p'_{in}, out_e) = \mathrm{TRANS}_e(e,V,L,tid)\\
&\qquad ((P_s,T_s,F_s), p_s, out_s) = \mathrm{TRANS}_s(s,V,L,tid)\\
&\qquad P = P_e \cup P_s \cup \{p_{in}\ \mathrm{fresh}(tid), p_e\ \mathrm{fresh}(tid)\}\\
&\qquad T = T_e \cup T_a \cup \{t_{head}\ \mathrm{fresh}(tid), t_t\ \mathrm{fresh}(tid), t_f\ \mathrm{fresh}(tid)\}\\
&\qquad F = F_e \cup F_s \cup \{(o_e,p_e) \mid o_e \in out_e\} \cup \{(o_s,p'_{in}) \mid o_s \in out_s\} \cup\\
&\qquad\qquad \{(p_{in},t_{head}), (t_{head},p'_{in}), (p_e,t_t), (p_e,t_f), (t_t,p_s)\}\\
&\mathrm{TRANS}_s(\texttt{atomic}\ s, V, L, tid) = \mathrm{TRANS}_s(s,V,L,tid)
\end{aligned}
$$

Fig. 5: Pseudocode of revised $TRANS(P)$

3.2 Enumerating the operations of movers

We state without proof the following property of our translation:

Claim. Consider a program $P \in$ SML such that $\vdash P : ok$, and a statement s in P, in thread T. Then the neighborhood of each transition created in TRANS(s) contains only a combination of places that are:

- locks (if the transition is a lock operation),
- variables (if it is a variable access), or
- in thread T as well (all other transitions).

Lemma 2. *Consider a program $P \in$ SML such that $\vdash P : ok$, and a statement s in P such that $\Gamma, \Sigma \vdash s : a, \Sigma'$ and $a \sqsubset A$. For an arbitrary trace, if there exists an event $e^T \in_\lambda$ TRANS(s) and there exists another event $f^{T'}$ where $T \neq T'$ such that that $e \prec f$ or $f \prec e$, then:*

1. *If $a \sqsubseteq R$, then either $\lambda(e) = t_x$ and $\Gamma(x) \in \Sigma$ or $\lambda(e) = t_{acq\ l}$; that is, e must be a race-free variable access or a lock-aquire.*
2. *If $a \sqsubseteq L$, then either $\lambda(e) = t_x$ and $\Gamma(x) \in \Sigma$ or $\lambda(e) = t_{rel\ l}$; that is, e must be a race-free variable access or a lock-release.*

Proof. By induction over the typing derivation of s:

- Case STMT-SKIP: $s = \texttt{skip}$. We must have $\lambda(e) = t^T_{skip}$, but the neighborhood of this transition is only places in thread T. Therefore we cannot have that $e \prec f$ or $f \prec e$; contradiction.
- Case STMT-ASSIGN: $s = x := exp$ and $\Gamma(x) \in \Sigma$. We could have $\lambda(e) = t^T_x$, which makes this case trivially true. If $\lambda(e) \neq t^T_x$, then $e \in_\lambda$ TRANS(exp). By the typing rule, we know $\Gamma, \Sigma \vdash exp : a'$, and that $(a'; B) \sqsubseteq A$, from which we can infer $a' \sqsubseteq A$. By induction over the typing derivation of exp:
  - Case EXP-CONST: $exp = c$. We must have $\lambda(e) = t^T_c$, but the neighborhood of this transition is only places in thread T. Therefore we cannot have that $e \prec f$ or $f \prec e$; contradiction.
  - Case EXP-READ: $exp = x$ and $\Gamma(x) \in \Sigma$. We must have $\lambda(e) = t^T_x$, which makes this case trivially true.
  - Case EXP-READ-RACE: $exp = x$ and $\Gamma(x) = \bullet$. By the typing rules, we have $\Gamma, \Sigma \vdash x : A, \Sigma$, so therefore we have a contradiction: $A \not\sqsubseteq A$.
  - Case EXP-PRIM: $exp = fn(e_1, \ldots, e_n)$. Examining the translation of primitive expressions, we know that $e \in_\lambda$ TRANS(e_i) for some i, since no other transitions are constructed. Further, by the definition of the $(\,;\,,\,)$ operator and the typing derivation, we know that for each subexpression e_i, its atomicity $a_i \sqsubseteq a'$, and by induction we are done.
- Case STMT-ASSIGN-RACE: $s = x := exp$ and $\Gamma(x) = \bullet$. We know that exp effect-checks, so let a' be its atomicity. By the typing rules, we have $\Gamma, \Sigma \vdash s : (a'; A), \Sigma$, so therefore we have a contradiction: $(a; A) \not\sqsubseteq A$.
- Case STMT-SEQ: $s = s_1; s_2$. Examining the translation of sequence statements, we see that $e \in_\lambda$ TRANS(s_1) or $e \in_\lambda$ TRANS(s_2), since no other transitions are constructed. Further, by the definition of the $(\,;\,)$ and the typing derivation, we know that the atomicities of either substatement must be at most equal to the atomicity of the whole statement. Therfore, by induction we are done.
- Case STMT-WHILE: $s = \texttt{while}\ exp\ body$. Examining the translation of while loops, we see that it constructs three transitions t^T_{head}, t^T_t and t^T_f, and recursively constructs TRANS(exp) and TRANS($body$). Further, we see that these three transitions are connected *only* to p'^T_{in}, p^T_e, p^T_s, and to whatever follows this loop in thread T; therefore $t^T_f I \lambda(f)$ and $t^T_t I \lambda f$, so therefore if $\lambda(e)$ were any of these three transitions, then $e \prec f$ and $f \prec e$ must be false. Therefore e cannot be any of these three events, so $e \in_\lambda$ TRANS(exp) or $e \in_\lambda$ TRANS($body$). Finally, from the typing derivation we can infer $\Gamma, \Sigma \vdash exp : a_{exp}$ and $\Gamma, \Sigma \vdash body : a_{body}, \Sigma$ and we know that $a = (a_{exp}; (a_{body}; a_{exp})^*)$. From the definition of the $(^*)$ operator we can infer that $a_{exp} \sqsubseteq a$ and similarly for a_{body}, and by induction we are done.
- Case STMT-IF: $s = \texttt{if}\ exp\ s_1\ s_2$. Examining the translation of if statements, we see that it constructs two transitions t^T_t and t^T_f, and recursively constructs TRANS(exp), TRANS(s_1), and TRANS(s_2). Further, we see that these two transitions are connected *only* to p^T_e, p_1 or p_2, all of which are in thread T, so if $\lambda(e)$ were either of these two transitions, then $e \prec f$ and $f \prec e$ must be false.

Therefore e cannot be either of these two transitions, so $e \in_\lambda$ TRANS(e), $e \in_\lambda$ TRANS(s_1) or $e \in_\lambda$ TRANS(s_2). Again using the definition of (;) we know that the atomicities of s_1 and s_2 are at most a, so by induction we are done.

– Case Stmt-Atomic: $s = \texttt{atomic}\ s'$. Examining the translation of atomic statements, we see that it recursively constructs TRANS(s'), and nothing else, and by induction we are done.
– Case Stmt-Acquire: $s = \texttt{acquire}\ l$. We must have that $\lambda(e) = t^T_{acq}$. If $a \sqsubseteq R$ then we are trivially true; else we are vacuously true.
– Case Stmt-Release: $s = \texttt{release}\ l$. We must have that $\lambda(e) = t^T_{rel}$. If $a \sqsubseteq L$ then we are trivially true; else we are vacuously true.

□

3.3 Sufficiency of our race detector

We need to show that the race analysis we encoded above is sufficient, and we do this in two steps:

Lemma 3. *Consider a progam $P \in$ Sml such that $\vdash P : ok$, and its translation* TRANS(P). *Consider a statement s in thread T such that $\Gamma, \Sigma \vdash s : a, \Sigma'$ occurs within the derivation of $\vdash P : ok$. For an arbitrary trace, consider an arbitrary event $e^T_e \in_\lambda$* END(s). *Suppose that*

$$\begin{aligned}\forall e^T_s \in_\lambda \mathrm{START}(s), e_s \in \mathrm{CURRENT}(s, e_e). \forall l \in \Sigma.&\\ \exists acq^T_l \in \mathcal{E}. acq_l \preceq e_s \wedge \lambda(acq_l) = t_{acq\ l} \wedge&\\ \nexists rel^T_l . acq_l \preceq rel_l \preceq e_s \wedge \lambda(rel_l) = t^T_{rel\ l}&\end{aligned}$$

Then

$$\begin{aligned}\forall f^T \succ e_e. \forall l \in \Sigma'. \exists acq^T_l.&\\ acq_l \preceq f \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge&\\ \nexists rel^T_l . acq_l \preceq rel_l \preceq f \wedge \lambda(rel_l) = t^T_{rel\ l}&\end{aligned}$$

In other words, if a set of locks Σ is held before a statement s executes in the Petri net, and s executes yielding a new set of locks Σ', then that set of locks truly is held after s executes.

Proof. By induction on $\Gamma, \Sigma \vdash s : a, \Sigma'$.

– Case Stmt-Atomic, Stmt-Assign, Stmt-Assign-Race, Stmt-Skip: None of these operations produce any events that acquire or release locks, except inductively on their substatements (if any). See the sequence case.
– Case Stmt-Seq: $s = (s_1; s_2)$. By assumption we know that e_e exists, and that END(s) = END(s_2). By induction, we can therefore say that if some Σ'' is held just prior to this execution of s_2, then Σ' will be held after it. By the construction of TRANS(s), we know that s_1 must have terminated

in order for s_2 to execute; therefore, $\exists e_1 \in \text{LAST}(s_1, e_e).e_1 \in_\lambda \text{END}(s_1)$. By induction, if Σ is held just before this execution of s_1, then Σ'' will be held just after it, giving the desired result.

– Case Stmt-If: Suppose $s = \texttt{if}\ e\ s_1\ s_2$. Then $TRANS(s)$ contains two transitions t_t and t_f, as well as the recursive constructions on the substatements. Trivially, t_t and t_f are not $t_{rel\ l}$. Also trivially, none of the transitions in TRANS(e) are $t_{rel\ l}$. This leaves only the substatements s_1 and s_2. We know that $\text{END}(s) = \text{END}(s_1) \cup \text{END}(s_2)$.

  Suppose $e_e \in_\lambda \text{END}(s_1)$ (the other case with s_2 is symmetric). We have $\Gamma, \Sigma \vdash s_1 : a_1, \Sigma$ from the typing derivation. Let e_s be as given in the assumption. For all $e_1^T \in_\lambda \text{START}(s_1)$ such that $e_s \preceq e_1 \preceq e_e$, we know that Σ is held before e_1 by the above reasoning. By induction, we know Σ truly is held after s_1, which in turn means it is held after s.

– Case Stmt-While: We have $s = \texttt{while}\ c\ b$. Examining TRANS(s), we see that $\text{END}(s) = \{t_f\}$ and $\text{START}(s) = \{t_{head}\}$. Therefore, $\exists e_c^T \in_\lambda \text{START}(c).e_s \preceq e_c \preceq e_e$. Now by the construction of TRANS(s), either $e_s \prec\!\!\cdot\, e_c$, or $\exists e_b \in_\lambda \text{END}(b).e_b \prec\!\!\cdot\, e_c$; this latter case corresponds to iterating the loop. Since we know that every event in a trace has a finite set of predecessors, we know that the while loop must have iterated only a finite number of times; we now use induction over the number of iterations. If $e_s \prec\!\!\cdot\, e_c$, then we are trivially done; Σ is unchanged by executing c. Otherwise, we effectively have a "sequence" $b; c$, the loop has iterated some n times. The same logic as for sequences shows that if Σ is held before the previous iteration of b, it must hold after this current iteration of c. Immediately preceding the execution of b is the transition t_t, which does not modify Σ, and preceding that are $n-1$ iterations of the while loop. By induction, we can conclude that if Σ is held before a while loop, it is held after $n-1$ iterations; the remaining arguments above show it is still held after n iterations.

– Case Stmt-Acquire: We have $\Gamma, \Sigma \vdash \texttt{acquire}(l) : R, \Sigma \cup \{\, l \,\}$. There is exactly one transition in TRANS(s), so $\text{START}(s) = \text{END}(s) = \{t^T_{acq\ l}\}$; moreover, $e_s = e_e$. For a lock $l' \in \Sigma'$:

  - If $l' = l$, then we trivially satisfy the condition above: we let $acq_l^T = e_s^T$.
  - If $l' \neq l$, then we know from the premise and from the observation that $e_s = e_e$ that

$$\begin{aligned}\exists acq_{l'}^T.acq_{l'} \preceq e_e \wedge \lambda(acq_{l'}) = t^T_{acq\ l'} \wedge \\ \nexists rel_{l'}^T.acq_{l'} \preceq rel_{l'} \preceq e_e \wedge \lambda(rel_{l'}) = t^T_{rel\ l'}\end{aligned}$$

    Since s doesn't touch l', we must have that this holds for all $f^T \succ\!\!\cdot\, e_e^T$, as desired.

– Case Stmt-Release: We have $\Gamma, \Sigma \vdash \texttt{release}(l) : L, \Sigma \setminus \{\, l \,\}$. By the same argument as above, since s doesn't touch l' for $l' \neq l$, we satisfy the condition. □

Lemma 4. *Consider a program $P \in$ SML such that $\vdash P : ok$. Then for every statement s such that $\Gamma, \Sigma \vdash s : a, \Sigma'$ appears in the derivation of $\vdash P : ok$, if*

$$\begin{aligned}\forall e^T \in_\lambda \mathrm{START}(s).\forall l \in \Sigma.\exists acq_l^T.\\ acq_l \preceq e \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge\\ \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t^T_{rel\ l}\end{aligned}$$

then for every statement (or expression) s' that appears within s, such that $\Gamma, \Sigma'' \vdash s' : a', \Sigma'''$ appears in the derivation of $\vdash P : ok$ we have the same property:

$$\begin{aligned}\forall e^T \in_\lambda \mathrm{START}(s').\forall l \in \Sigma''.\exists acq_l^T.\\ acq_l \preceq e \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge\\ \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t^T_{rel\ l}\end{aligned}$$

Proof. Expressions do not change locksets, so if a lockset Σ holds at the beginning of an expression *exp*, it holds at the beginning of every subexpression of *exp*. Therefore we need only examine statements. By induction on the structure of s:

- Case $s =$ `skip`: vacuously true.
- Case $s = x := exp$: Looking at TRANS(s), we see that START(s) = START(exp) and $\Sigma'' = \Sigma$, so by induction we are done.
- Case $s =$ `if` $c\ s_1\ s_2$: Looking at TRANS(s), we see that START(s) = START(c), so Σ holds at the beginning of c. Since the transitions t_t and t_f do not modify locks, we know that Σ holds at the beginning of s_1 and s_2 as well; by induction, we are done.
- Case $s =$ `acquire` l or $s =$ `release` l: vacuously true.
- Case $s =$ `atomic` s': Since START(s) = START(s'), by induction we are done.
- Case $s = (s_1; s_2)$: Since START(s) = START(s_1), by induction we are done with s_1. By the typing rules, we must have that $\Gamma, \Sigma \vdash s_1 : a_1, \Sigma_1$ and $\Gamma, \Sigma_1 \vdash s_2 : a_2, \Sigma_2$. If an event e_2 exists in START(s_2), then s_1 must have terminated. We can pick any $e_1 \in$ LAST(s_1, e_2), which implies $e_1 \prec e_2$, and by Lemma 3, we know that Σ_1 is held just after e_1, which is to say, just before e_2, so by induction we are done with s_2.
- Case $s =$ `while` $c\ b$: The transition t_{head} obviously does not modify any locks, so we need a second level of induction here, since the condition c can execute multiple times. We must show that if the loop has run for a finite number of iterations (n), then on the $n + 1$ execution of c, the locks Σ are held just prior to the start of c. It is clear that the set $\{e \in_\lambda \mathrm{START}(c)\} = \{e_0, e_1, \ldots\}$ is the totally ordered set of events corresponding to each iteration of c: event e_n occurs when the loop has executed n times. Let $f_n \prec e_n$ be the event just prior to e_n in the same thread. By induction on n:
  - Case $n = 0$: The condition has not executed at all, therefore $\lambda(f_0) = t_{head}$, and so the same locks Σ are held before e_0 as were held at the start of the loop.

- Case $n > 0$: Suppose the loop has executed n times, and we're about to execute e_n. Then $f_n \in_\lambda \text{END}(b)$ and $f_n \prec e_n$. By induction, the locks Σ are held before e_{n-1}. Then after c executes, the locks are still held (because expressions don't change locksets), after t_t executes the locks are still held (since t_t is not a lock operation), and by Lemma 3, some lockset Σ' is held after f_n, and by the Stmt-While rule, we know that $\Sigma' = \Sigma$. Therefore, the lockset Σ is held just prior to e_n as well, as required. (We know that executing b for the nth time terminates, because event e_n exists.)

□

The key corollary to this is

Lemma 5. *Consider a program $P \in$ Sml such that $\vdash P : ok$. Then for every statement s such that $\Gamma, \Sigma \vdash s : a, \Sigma'$ appears in the derivation of $\vdash P : ok$, we know that*

$$\begin{aligned}&\forall e^T \in_\lambda \text{START}(s).\forall l \in \Sigma.\exists acq_l^T.\\&\qquad acq_l \preceq e \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge\\&\qquad \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t^T_{rel\ l}\end{aligned}$$

Proof. We know $P = s_1||s_2||\ldots||s_n$. For every statement s_i, we know $\Gamma, \emptyset \vdash s_i : a_i, \Sigma'$, and we know all threads do start with no locks held in the Petri net, we can apply the above lemma. □

We can therefore conclude

Lemma 6. *Our race detector is sufficient.*

Proof. Consider two accesses to variable x in different threads that are determined to be race-free, as required by the definition of sufficient race detectors; let s and t be the primitive statements or expressions in which these accesses occur, and T and T' be their threads respectively. By Lemma 5, we know that

$$\begin{aligned}&\forall e^T \in_\lambda \text{START}(s).\forall l \in \Sigma.\exists acq_l^T.\\&\qquad acq_l \preceq e \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge\\&\qquad \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t^T_{rel\ l}\end{aligned}$$

(and similarly for statement t) and in particular, since the accesses are race-free,

$$\begin{aligned}&\forall e^T \in_\lambda \text{START}(s).\exists acq_{\Gamma(x)}^T.\\&\qquad acq_{\Gamma(x)} \preceq e \wedge \lambda(acq_{\Gamma(x)}) = t^T_{acq\ \Gamma(x)} \wedge\\&\qquad \nexists rel_{\Gamma(x)}^T.acq_{\Gamma(x)} \preceq rel_{\Gamma(x)} \preceq e \wedge \lambda(rel_{\Gamma(x)}) = t^T_{rel\ \Gamma(x)}\end{aligned}$$

We can conclude that $acq^T_{\Gamma(x)} \preceq e^T_s$, for the specific event $\lambda(e^T_s) = t^T_x$ that accesses x in statement s, and similarly $acq^{T'}_{\Gamma(x)} \preceq e^{T'}_t$ for the event $\lambda(e^{T'}_t) =$

$t_x^{T'}$ that accesses x in statement t. Assume that $e_s^T \preceq e_t^{T'}$ — they must be ordered since they access the same variable (the opposite ordering is symmetric). Therefore $acq_{\Gamma(x)}^T \preceq e_s^T \preceq e_t^{T'}$. We know that $acq_{\Gamma(x)}^T \preceq acq_{\Gamma(x)}^{T'}$ or vice versa, since they access the same lock. We proceed by cases to determine when $acq_{\Gamma(x)}^{T'}$ can occur (a symmetric argument holds when thread T' starts first):

1. $acq_{\Gamma(x)}^{T'} \preceq acq_{\Gamma(x)}^T \preceq e_s^T \preceq e_t^{T'}$. This is a contradiction: from the line above we know that the lock $\Gamma(x)$ is continually held from $acq_{\Gamma(x)}^{T'}$ until $e_t^{T'}$ (since no release event happens between these two events), therefore the place $\Gamma(x)_{open}$ is unmarked, so therefore $acq_{\Gamma(x)}^T$ cannot happen here.
2. $acq_{\Gamma(x)}^T \preceq acq_{\Gamma(x)}^{T'} \preceq e_s^T \preceq e_t^{T'}$. This is a contradiction for the same reason: we know that the lock $\Gamma(x)$ is continually held from $acq_{\Gamma(x)}^T$ until e_t^T, so therefore $acq_{\Gamma(x)}^{T'}$ cannot happen here.
3. $acq_{\Gamma(x)}^T \preceq e_s^T \preceq acq_{\Gamma(x)}^{T'} \preceq e_t^{T'}$. This is the interesting case. It obeys all the constraints we know so far, however, the lock $\Gamma(x)$ is still held by thread T until released. The only way for these four events to occur in this order is for there to be a *fifth* event in thread T that releases the lock:

$$acq_{\Gamma(x)}^T \preceq e_s^T \preceq rel_{\Gamma(x)}^T \preceq acq_{\Gamma(x)}^{T'} \preceq e_t^{T'}$$

   where $\lambda(rel_{\Gamma(x)}^T) = t_{rel\ \Gamma(x)}^T$. Now we have precisely the events required by our definition of sufficient race conditions: the last four events above.
4. $acq_{\Gamma(x)}^T \preceq e_s^T \preceq e_t^{T'} \preceq acq_{\Gamma(x)}^{T'}$. Again we have a contradiction: we know $acq_{\Gamma(x)}^{T'} \preceq e_t^{T'}$.

Therefore, whenever our race detector concludes that two accesses are race-free, we are guaranteed two events, one in each thread, that happen between the two variable accesses, as required by our definition of sufficient race detectors. □

3.4 Causal atomicity of sequenced statements

Now we have a powerful lemma, which says that if we have a statement (or expression) that effect-checks as atomic, and its translation to a Petri net can be decomposed into two subnets, both of which are known to be causally atomic, then the combined net is also causally atomic. Intuitively, we show this by noting that if there were an event that witnesses the non-causal-atomicity of the full statement, it must be causally between the two subnets, and then show that even that causal ordering is impossible.

Lemma 7. *Assume a program $P \in$ SML such that $\vdash P : ok$. Consider two statements (or expressions) s_1 and s_2 found in P, both in the same thread T, and let $N_1 = \text{TRANS}(s_1)$ and $N_2 = \text{TRANS}(s_2)$ be subnets of $\text{TRANS}(P)$ that result from the translation of the two statements. Assume that:*

1. *$\Gamma, \Sigma \vdash s_1 : a_1, \Sigma'$ and $\Gamma, \Sigma' \vdash s_2 : a_2, \Sigma''$, and $(a_1; a_2) \sqsubseteq A$.*

2. *N_1 and N_2 are causally atomic within* TRANS(P).
3. $\forall t_1 \in \text{END}(s_1).\exists t_2 \in \text{START}(s_2).t_1^\bullet \cap {}^\bullet t_2 \neq \emptyset$ *in the full Petri net, that is, that N_2 is "immediately after" N_1 in* TRANS(P).

Then $N_1 \cup N_2$ is causally atomic.

Proof. By inspecting the operator (;) and by assumption 1, we see that $a_1 \sqsubseteq R$ or $a_2 \sqsubseteq L$ (or both). We have that s_1 and s_2 are causally atomic by assumption 2. From the definition of causal atomicity, we therefore know that for any possible trace,

$$\nexists u_2^T \in_\lambda \text{TRANS}(s_1), f^{T'} \in \mathcal{E}.(u_{start}^T \preceq f^{T'} \preceq u_2^T) \wedge \neg(u_{start}^T \preceq u_{end}^T \preceq u_2^T)$$
$$\nexists v_2^T \in_\lambda \text{TRANS}(s_2), g^{T''} \in \mathcal{E}.(v_{start}^T \preceq g^{T''} \preceq v_2^T) \wedge \neg(v_{start}^T \preceq v_{end}^T \preceq v_2^T)$$

where $u_{start}^T \in_\lambda \text{START}(s_1)$ and $u_{end}^T \in_\lambda \text{END}(s_1)$ (resp. v_{start}^T and v_{end}^T) correspond to transitions in the START and END sets in the translation of s_1 (resp. s_2) and event $f^{T'}$ (resp. $g^{T''}$) is not in the same thread T. We therefore want to show that for every possible trace,

$$\nexists e_2^T \in_\lambda \text{TRANS}(s_1) \cup \text{TRANS}(s_2), h^{T'''} \in \mathcal{E}.$$
$$(e_{start} \preceq h \preceq e_2) \wedge \nexists e_{end}^T \in_\lambda \text{END}(s_2).(e_{start} \preceq e_{end} \preceq e_2)$$

where as above, $e_{start}^T \in_\lambda \text{START}(s_1)$ and $e_{end}^T \in_\lambda \text{END}(s_2)$, and $h^{T'''}$ is an event in some thread $T''' \neq T$.

We proceed by contradiction. Suppose there existed some trace in which such an event $h^{T'''}$ did exist (and an event e_2^T that follows it). Such an event would satisfy

$$(e_{start} \preceq h \preceq e_2) \wedge \neg(e_{start} \preceq e_{end} \preceq e_2)$$

We know that LAST(s_1, h) is not empty, because by assumption, $e_{start} \preceq h$, so it must be that either $e_{start} \in \text{LAST}(s_1, h)$ or else some other event e_L^T exists such that $e_{start} \preceq e_L \preceq h$, and $e_L \in \text{LAST}(s_1, h)$.

We know that FIRST(s_1, h) is empty, because otherwise we could violate the atomicity of TRANS(s_1):

$$\exists e_F^T \in \text{FIRST}(s_1, h).e_{start} \preceq h \preceq e_F$$

which is a contradiction. So h interacts with something in LAST(s_1, h) and nothing after it in TRANS(s_1).

We know that e_2 must be in TRANS(s_2), since FIRST(s_1, h) is empty, and no other events exist besides those in TRANS(s_1) and TRANS(s_2). Therefore, we know that either $e_2 \in \text{FIRST}(s_2, h)$ or else some other event e_F^T exists such that $h \preceq e_F \preceq e_2$, and hence FIRST($s_2, h$) is not empty.

We know that LAST(s_2, h) is empty, because otherwise we could violate the atomicity of TRANS(s_2):

$$\exists e_L^T \in \text{LAST}(s_2, h).e_L \preceq h \preceq e_2$$

which is a contradiction. So h interacts with something in $\text{LAST}(s_2, h)$ and nothing before it in $\text{TRANS}(s_2)$.

Therefore it must be that

$$\exists u^T \in \text{LAST}(s_1, h), v^T \in \text{FIRST}(s_2, h).u \preceq h \preceq v$$

In other words, if an event h exists at all, it must happen between the last interfering event in $\text{TRANS}(s_1)$ and the first interfering event in $\text{TRANS}(s_2)$ (based on the two sets above that are non-empty). Additionally

$$\nexists w \in_\lambda \text{TRANS}(s_1) \cup \text{TRANS}(s_2). \\ \forall u^T \in \text{LAST}(s_1, h), v^T \in \text{FIRST}(s_2, h).u \preceq w \preceq v$$

In other words, there are no other interfering events in the translations of the two statements between those identified before (based on the two sets above that are empty).

It is possible that h is not unique, that is, there might be multiple events causally between u and v (for any pair of events u and v as above). Therefore, define events $h_1^{T'''}$ and $h_2^{T'''}$, not necessarily distinct from $h^{T'''}$, such that

$$u^T \prec h_1^{T'''} \preceq h^{T'''} \preceq h_2^{T'''} \prec v^T$$

Now we have our contradiction. Suppose s_1 is a right-mover. Then the event u must either be a lock acquire or a race-free variable access (by Lemma 2 above), and so must event h_1 (since $u \prec h_1$, which can only happen if both events access the same resource).

If both events are lock acquires of some lock l, then we have an immediate contradiction, since it is not possible for two threads to acquire the same lock simultaneously. In our translation into Petri nets, a lock-acquire operation moves a mark from l_{open} to l_T. Any other lock-acquire operations on the same lock l are not enabled until the lock is released and the mark is restored to l_{open}. However, we have that $u \prec h_1$, so no lock-release event occurs before h_1 supposedly happens. (A similar contradiction exists if h_1 is a lock release; thread T''' can't release a lock held by thread T.)

If both events access a variable (and clearly one must be a variable write, or else there is no conflict) then we appeal to the race-freedom of the access. We assumed the race detector was sound, so therefore both events must be race free. We also assumed that it was sufficient, so we know some lock l must be held for all accesses to this variable, or else there may be a race condition. So there must be some event in thread T which acquires lock l, and similarly there must be some event in thread T''' that acquires the same lock. Since we assumed that $u \prec h_1$, and u doesn't release the lock (it's a variable access, and does nothing to locks), h_1 must happen after the lock is released. We assumed that s_1 is a right mover, which means it cannot release the lock. So any further events that interact with lock l must occur in $\text{TRANS}(s_2)$ (by assumption 3, since $\text{TRANS}(s_2)$ is "immediately after" $\text{TRANS}(s_1)$ in the overall net—no other events in thread

T can intervene). Clearly, the first such event must be to release the lock (since it is currently held), and this could happen before or after event v. But the lock must be held for event v, since it is a race-free access, so if the lock *were* released, it must be reacquired before event v. But this is not atomic, which contradicts our assumption that s_2 effect-checked as atomic. Therefore, the lock l must be continuously held between u and v. We must have therefore that $v \preceq h_1$, but this contradicts our assumption that $h \preceq v$. We can conclude that if s_1 is a right-mover, then no event h can occur during s_1, between s_1 and s_2, or during s_2, and therefore that the union $N_1 \cup N_2$ is causally atomic.

A symmetric argument holds when s_2 is a left-mover. □

3.5 Causal atomicity encompasses effect-based atomicity

We can now state the main theorem of this section, which shows that causal atomicity is at least as strong as effect-based atomicity.

Theorem 1. *For every program $P \in$ SML where $\vdash P : ok$, then all* `atomic` *blocks in P are causally atomic when translated into Petri nets.*

Proof. Consider an arbitrary expression $e \in P$ within an arbitrary atomic block. We assume that all of P effect-checks, including all `atomic` blocks. We wish to show that if $\Gamma, \Sigma \vdash e : a$ and $a \sqsubseteq A$, then TRANS(e) is causally atomic in the full net resulting from the translation of P. We proceed by induction on the typing derivation:

- Case Exp-Const: We have $e = c$, which translates to a single transition, and therefore we are trivially atomic.
- Case Exp-Var and Exp-Var-Race: We have $e = x$, which translates to a single transition, and therefore we are trivially atomic.
- Case Exp-Prim: We have $e = fn(e_1, \ldots, e_n)$. The translation of this yields several subnets, one for each subexpression e_i. We know that $\Gamma, \Sigma \vdash e : a$ and $a \sqsubseteq A$, so we know that $(a_1; \ldots; a_n) \sqsubseteq A$, where a_i is the atomicity of subexpression e_1. Finally, each subexpression is connected in succession to the following one. This meets all the criteria for Lemma 7, and so by $n-1$ applications of the lemma, we are done.

Consider an arbitrary statement $s \in P$ within an arbitrary atomic block. We assume that all of P effect-checks, including all `atomic` blocks. We wish to show that if $\Gamma, \Sigma \vdash s : a, \Sigma'$ and $a \sqsubseteq A$, then TRANS(s) is causally atomic in the full net resulting from the translation of P. We proceed by induction on the typing derivation:

- Case Stmt-Seq: We have $s = s_1; s_2$, and we know that $\Gamma, \Sigma \vdash s : a, \Sigma'$, and $a \sqsubseteq A$. We therefore know that $\Gamma, \Sigma \vdash s_1 : a_1, \Sigma''$ and $\Gamma, \Sigma'' \vdash s_2 : a_2, \Sigma'$ and that $(a_1; a_2) = a$. This meets all the criteria for Lemma 7, and so we are done.

– Case Stmt-If: $s = \texttt{if}\ e\ s_1\ s_2$. The translation of s decomposes into three subnets, one per subexpression, as well as two extra transitions t_t and t_f indicating which branch of the `if` to take. By induction, we assume that each of these subnets is causally atomic. It is trivial to see that "grouping" t_t with TRANS(s_1) and t_f with TRANS(s_2) does not affect their causal atomicity: since these transitions are immediately before only the entry points of TRANS(s_1) and TRANS(s_2), any event f that could witness the non-atomicity of the grouping would witness the non-atomicity of the subexpressions themselves. By assumption, $\Gamma, \Sigma \vdash s : a, \Sigma$ and $a \sqsubseteq A$. Following the inference rule, we have $\Gamma, \Sigma \vdash e : a_e$ and $\Gamma, \Sigma \vdash s_i : a_i, \Sigma$. We therefore know that $(a_e; a_1) \sqsubseteq a$ and $(a_e; a_2) \sqsubseteq a$. This meets all the requirements for Lemma 7, so we apply it twice, with e and s_1 and then with e and s_2, and so we are done.
– Case Stmt-Loop: $s = \texttt{while}\ e\ s'$. The translation of s decomposes into two subnets, one for the guard and one for the body, as well as two extra transitions t_t and t_f indicating whether to loop or to exit. Inductively we assume that each subnet is causally atomic. Again we can group t_t with TRANS(s') without affecting its atomicity. Using the typing rule, we see that a loop's type is equivalent to the transitive closure of its body's type when repeated indefinitely. Examining the closure operation, we see that $a^* = (a; a)$ for all atomicities. Therefore the atomicity of the loop is $a = (a_e; a_{s'}; a_e; a_{s'}; a_e)$. By examination, for $a \sqsubseteq A$ we must have $a_e \sqsubset R \wedge a_s = B$ or $a_e = B \wedge a_s \sqsubset L$. We apply Lemma 7 twice: first with e and s', and the second time with s' and e, indicating that no atomicity violation can occur between the condition and the body or between the body and the return to the condition. Finally, since the union of the two subexpressions is causally atomic, we can group the t_f transition to it without changing the atomicity, showing that the whole loop statement is causally atomic.
– Case Stmt-Acquire, Stmt-Release, Stmt-Skip: These each translate to a single transition, and therefore we are trivially atomic.
– Case Stmt-Atomic: $s = \texttt{atomic}\ s'$. But TRANS($\texttt{atomic}\ s'$) = TRANS(s'), and by induction we are done.
– Case Stmt-Assign, Stmt-Assign-Race: $s = x := exp$. The translation of this expression decomposes into two subnets: the TRANS(exp), and a transition t_x^T that accesses the variable x. We know that TRANS(e) is causally atomic. Therefore, if there exists an event $f^{T'}$ that witnesses the non-atomicity of TRANS(s), it must satisfy

$$\exists e_1^T \in_\lambda \text{START}(s), e_2^T \in_\lambda \mathcal{E}. \\ (e_1 \preceq f \preceq e_2) \wedge (\nexists e_{end}^T \in_\lambda \text{END}(s). e_1 \preceq e_{end} \preceq e_2)$$

By examining TRANS(s), we see that END(s) = $\{t_x^T\}$ and START(s) = START(exp). Moreover, we know that $e_2 \notin \text{CURRENT}(exp, e_1)$, since if it were, then $\nexists e_{end} \in \text{END}(exp). e_{end} \preceq e_2$, and then f would witness the non-atomicity of TRANS(exp), which is impossible by assumption. Therefore

we must have $\text{CURRENT}(s, e_1) \setminus \text{CURRENT}(exp, e_1) = \{e_2\}$. But then $\exists e_{end} \in \text{END}(s).e_{end} \preceq e_2$, which contradicts our assumption about f.

□

4 Pure-Sml: distilling atomicity from impure sources

Syntax of Pure-Sml

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

Fig. 6: The language Pure-Sml, with a few slight cosmetic changes to more closely match Cat and Cap

4.1 Language

Figure 6 defines the language Pure-Sml. Compared to Sml, the key addition is a new `pure` statement that indicates the contained code should be side-effect free. To simplify the analysis, `while` loops have become infinite `loop`s; to escape them, a `break` statement is introduced that jumps to the end of the nearest enclosing `block` statement. (Hence, a `while` loop now is written as `block loop`.)

The distinction between $StableVar$s and $UnstableVar$s now becomes important: $UnstableVar$s may be written to within a `pure` block, but they can take on any (non-deterministic) value. This is the essence of the abtract atomicity which is presented in Cap.

4.2 Effect-system related definitions

We need to provide a purity analysis for Pure-Sml, which we can then translate to widgets in Petri nets. Looking at the purity effect system in Cap, we see that all functions used in the supposedly pure expression must be pure, and that the expression itself must be pure. Further, pure is defined to mean that the expression only writes to unstable variables, and terminates with exactly the same set of locks held as it help at its beginning. Moreover, looking at the abstract semantics of Cap (against which the authors define their notion of atomicity), we

see that pure blocks can be skipped altogether, and the expressions thus skipped can evaluate to any value consistent with evaluating the skipped expression in an undefined context. (This is an optimization that permits constant expressions to be relied upon outside a pure block.)

For Pure-Sml, this simplifies substantially: we have no functions, and our programs consist of statements, so pure statements need not return any value at all. Therefore, to track purity, we have only to track locksets and variable accesses within pure blocks. The effect system that achieves this (as well as providing atomicity and race-detection analyses) is shown in figure 7. As in Cap, we extend the atomicity annotations to a pair of basic atomicities $a \uparrow b$, where the first component indicates the atomicity of the statement if it terminates normally (i.e., without `break`ing) and the second component indicates the atomicity under abrupt termination (i.e. via `break`). The new atomicity $\bot$ indicates that the statement cannot terminate in the given mode; for instance, expressions never terminate abruptly. The rule Stmt-Block shows that `block` statements always terminate normally, with atomicity approximated by the join of normal and abrupt atomicities. Stmt-Loop shows that loops never terminate normally, but may run an unbounded number of times (a^*) before terminating abruptly (b). Finally, variable writes now check that the variable x is in the set of modifiable variables X; initially X is the set of all variables, but is limited to $UnstableVar$s inside `pure` blocks. Moreover, Stmt-Pure ensures that pure blocks terminate with the same lockset with which they began. As in Cap, we enforce a syntactic restriction on `pure` blocks, and prohibit them from being syntactically nested.

4.3 Petri-net related definitions

Constructing this effect system in Petri nets is accomplished by modifying the translation function shown before. The important changes to the translation are shown graphically in figure 8, and defined explicitly in figure 11. Five statement forms are crucial: variable accesses, lock accesses, `pure` statements, `block` statements, `break` statements; we construct each of these in turn.

Handling variable accesses uses the same construction as in Sml; the differences lie in the coloring rules. Informally, when checking for purity violations each mark contains a color indicating whether it is inside a pure block that is ok, i.e. still pure (o), or inside a pure block that has gone "bad" and done an impure write (b). Colors start at o upon entry to a pure block, and change to b upon stable variable writes, and stay unchanged otherwise. It is an error to exit a pure block normally if the color is b; a break statement must be used instead. We defer detecting purity violations by writes until exiting the `pure` block, because we cannot know whether the writes will be retroactively permitted by a `break` statement until we reach the end of the block. We will return to these coloring rules more formally later.

Locks are handled slightly differently than in the basic causal atomicity case. Here, not only do we have a place L for every lock, but we also have two place L_i-*held* and L_i-*other*, that indicate for each thread i whether it is held or not by

Atomicity effect system of Pure-Sml

$a, b \in Atomicity = \{\bot(\text{will not terminate}), B, L, R, A, \top(\text{not atomic})\}$

$\boxed{\Gamma \vdash e : a}$

[Exp-Const]
$$\frac{}{\Gamma \vdash c : B}$$

[Exp-Var]
$$\frac{\Gamma(x) \neq \bullet}{\Gamma \vdash x : B}$$

[Exp-Var-Race]
$$\frac{\Gamma(x) = \bullet}{\Gamma \vdash x : A}$$

[Exp-Prim]
$$\frac{\Gamma \vdash e_i : a_i}{\Gamma \vdash fn(e_1, \ldots, e_n) : (a_1; \ldots; a_n; \Gamma(p))}$$

$\boxed{\Gamma \vdash s : a \uparrow b}$

[Stmt-Pure]
$$\frac{\Gamma \vdash s : a \uparrow b \qquad a \sqsubseteq A}{\Gamma \vdash \texttt{pure}\ s : B \uparrow b}$$

[Stmt-Skip]
$$\frac{}{\Gamma \vdash \texttt{skip} : B \uparrow \bot}$$

[Stmt-Seq]
$$\frac{\Gamma \vdash s_1 : a_1 \uparrow b_1 \qquad \Gamma \vdash s_2 : a_2 \uparrow b_2}{\Gamma \vdash s_1; s_2 : (a_1; a_2) \uparrow (b_1 \sqcup (a_1; b_2))}$$

[Stmt-Loop]
$$\frac{\Gamma \vdash s : a \uparrow b}{\Gamma \vdash \texttt{loop}\ s : \bot \uparrow (a^*; b)}$$

[Stmt-Assign]
$$\frac{\Gamma \vdash e : a \qquad \Gamma(x) \neq \bullet}{\Gamma \vdash x := e : (a; B) \uparrow \bot}$$

[Stmt-Assign-Race]
$$\frac{\Gamma \vdash e : a \qquad \Gamma(x) = \bullet}{\Gamma \vdash x := e : (a; A) \uparrow \bot}$$

[Stmt-Atomic]
$$\frac{\Gamma \vdash s : a \uparrow b \qquad a, b \sqsubseteq A}{\Gamma \vdash \texttt{atomic}\ s : a \uparrow b}$$

[Stmt-If]
$$\frac{\Gamma \vdash e : a_e \qquad \Gamma \vdash s_1 : a_1 \uparrow b_1 \qquad \Gamma \vdash s_2 : a_2 \uparrow b_2}{\Gamma \vdash \texttt{if}\ e\ s_1\ s_2 : (a_e; (a_1 \sqcup a_2)) \uparrow (a_e; (b_1 \sqcup b_2))}$$

[Stmt-Acquire]
$$\frac{}{\Gamma \vdash \texttt{acquire}\ l : R \uparrow \bot}$$

[Stmt-Release]
$$\frac{}{\Gamma \vdash \texttt{release}\ l : L \uparrow \bot}$$

[Stmt-Block]
$$\frac{\Gamma \vdash s : a \uparrow b}{\Gamma \vdash \texttt{block}\ s : (a \sqcup b) \uparrow \bot}$$

[Stmt-Break]
$$\frac{}{\Gamma \vdash \texttt{break} : \bot \uparrow B}$$

Fig. 7: The effect systems for atomicity, race detection, and purity.

Race-detection effect system of PURE-SML

$\boxed{\Gamma, \Sigma \vdash e : \text{ok}}$

[EXP-CONST]
$$\frac{}{\Gamma, \Sigma \vdash c : \text{ok}}$$

[EXP-VAR]
$$\frac{\Gamma(x) \in \Sigma}{\Gamma, \Sigma \vdash x : \text{ok}}$$

[EXP-VAR-RACE]
$$\frac{\Gamma(x) = \bullet}{\Gamma, \Sigma \vdash x : \text{ok}}$$

[EXP-PRIM]
$$\frac{\Gamma, \Sigma \vdash e_i : \text{ok}}{\Gamma, \Sigma \vdash fn(e_1, \ldots, e_n) : \text{ok}}$$

$\boxed{\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''}$

[STMT-PURE]
$$\frac{\Gamma, \Sigma \vdash s : \Sigma \uparrow \Sigma'}{\Gamma, \Sigma \vdash \texttt{pure}\ s : \Sigma \uparrow \Sigma'}$$

[STMT-BREAK]
$$\frac{}{\Gamma, \Sigma \vdash \texttt{break} : \Sigma' \uparrow \Sigma}$$

[STMT-ASSIGN]
$$\frac{\Gamma(x) \in \Sigma \qquad \Gamma, \Sigma \vdash e : \text{ok}}{\Gamma, \Sigma \vdash x := e : \Sigma \uparrow \Sigma'}$$

[STMT-ASSIGN-RACE]
$$\frac{\Gamma(x) = \bullet \qquad \Gamma, \Sigma \vdash e : \text{ok}}{\Gamma, \Sigma \vdash x := e : \Sigma \uparrow \Sigma'}$$

[STMT-SEQ]
$$\frac{\Gamma, \Sigma \vdash s_1 : \Sigma_1 \uparrow \Sigma_1' \qquad \Gamma, \Sigma_1 \vdash s_2 : \Sigma_2 \uparrow \Sigma_2'}{\Gamma, \Sigma \vdash s_1; s_2 : \Sigma_2 \uparrow \Sigma_1' \cap \Sigma_2'}$$

[STMT-SKIP]
$$\frac{}{\Gamma, \Sigma \vdash \texttt{skip} : \Sigma \uparrow \Sigma'}$$

[STMT-LOOP]
$$\frac{\Gamma, \Sigma \vdash s : \Sigma \uparrow \Sigma'}{\Gamma, \Sigma \vdash \texttt{loop}\ s : \Sigma'' \uparrow \Sigma'}$$

[STMT-ATOMIC]
$$\frac{\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''}{\Gamma, \Sigma \vdash \texttt{atomic}\ \ s : \Sigma' \uparrow \Sigma''}$$

[STMT-IF]
$$\frac{\Gamma, \Sigma \vdash s_1 : \Sigma \uparrow \Sigma' \qquad \Gamma, \Sigma \vdash s_2 : \Sigma \uparrow \Sigma' \qquad \Gamma, \Sigma \vdash e : \text{ok}}{\Gamma, \Sigma \vdash \texttt{if}\ e\ s_1\ s_2 : \Sigma \uparrow \Sigma'}$$

[STMT-BLOCK]
$$\frac{\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''}{\Gamma, \Sigma \vdash \texttt{block}\ s : \Sigma' \cap \Sigma'' \uparrow \Sigma'''}$$

[STMT-ACQUIRE]
$$\frac{l \notin \Sigma}{\Gamma, \Sigma \vdash \texttt{acquire}\ l : \Sigma \cup \{\, l \,\} \uparrow \Sigma'}$$

[STMT-RELEASE]
$$\frac{l \in \Sigma}{\Gamma, \Sigma \vdash \texttt{release}\ l : \Sigma \setminus \{\, l \,\} \uparrow \Sigma'}$$

Fig. 7: The effect systems for atomicity, race detection, and purity.

Purity effect systems of Pure-Sml

$p \in \mathit{Purity} = \{\bot(\text{must be pure on normal termination}), \top(\text{may be impure})\}$

$\boxed{\Gamma, X, p, \Sigma \vdash s : p', \Sigma'}$

[Stmt-Pure]
$$\frac{\Gamma, X \cap \mathit{UnstableVar}, \bot, \Sigma \vdash s : \bot, \Sigma}{\Gamma, X, p, \Sigma \vdash \texttt{pure}\ s : p, \Sigma}$$

[Stmt-Break]
$$\frac{}{\Gamma, X, p, \Sigma \vdash \texttt{break} : \bot, \Sigma'}$$

[Stmt-Assign]
$$\frac{\Gamma(x) \in \Sigma \qquad x \in X}{\Gamma, X, p, \Sigma \vdash x := e : \top, \Sigma}$$

[Stmt-Assign-Race]
$$\frac{\Gamma(x) = \bullet \qquad x \in X}{\Gamma, X, p, \Sigma \vdash x := e : \top, \Sigma}$$

[Stmt-Seq]
$$\frac{\Gamma, X, p, \Sigma \vdash s_1 : p_1, \Sigma_1 \qquad \Gamma, X, p_1, \Sigma_1 \vdash s_2 : p_2, \Sigma_2}{\Gamma, X, p, \Sigma \vdash s_1; s_2 : p_2, \Sigma_2}$$

[Stmt-Skip]
$$\frac{}{\Gamma, X, p, \Sigma \vdash \texttt{skip} : p, \Sigma}$$

[Stmt-Loop]
$$\frac{\Gamma, X, p, \Sigma \vdash s : p', \Sigma}{\Gamma, X, p, \Sigma \vdash \texttt{loop}\ e\ s : p', \Sigma}$$

[Stmt-Atomic]
$$\frac{\Gamma, X, p, \Sigma \vdash s : p', \Sigma'}{\Gamma, X, p, \Sigma \vdash \texttt{atomic}\ \ s : p', \Sigma'}$$

[Stmt-If]
$$\frac{\Gamma, X, p, \Sigma \vdash s_1 : p_1, \Sigma \qquad \Gamma, X, p, \Sigma \vdash s_2 : p_2, \Sigma}{\Gamma, X, p, \Sigma \vdash \texttt{if}\ e\ s_1\ \ s_2 : p_1 \sqcup p_2, \Sigma}$$

[Stmt-Block]
$$\frac{\Gamma, X, p, \Sigma \vdash s : p', \Sigma'}{\Gamma, X, p, \Sigma \vdash \texttt{block}\ s : p', \Sigma'}$$

[Stmt-Acquire]
$$\frac{l \notin \Sigma}{\Gamma, X, p, \Sigma \vdash \texttt{acquire}\ l : p, \Sigma \cup \{\, l \,\}}$$

[Stmt-Release]
$$\frac{l \in \Sigma}{\Gamma, X, p, \Sigma \vdash \texttt{release}\ l : p, \Sigma \setminus \{\, l \,\}}$$

Valid programs of Pure-Sml

$\boxed{\vdash P : ok}$

[Prog-OK]
$$\frac{\Gamma, \emptyset \vdash s_i : \Sigma_i \uparrow \Sigma_i' \qquad \Gamma \vdash s_i : a_i \uparrow b_i \qquad \Gamma, \mathit{Var}, \bot, \emptyset \vdash s_i : p_i, \Sigma_i \qquad 1 \le i \le n}{\vdash s_1 || \cdots || s_n : ok}$$

Fig. 7: The effect systems for atomicity, race detection, and purity.

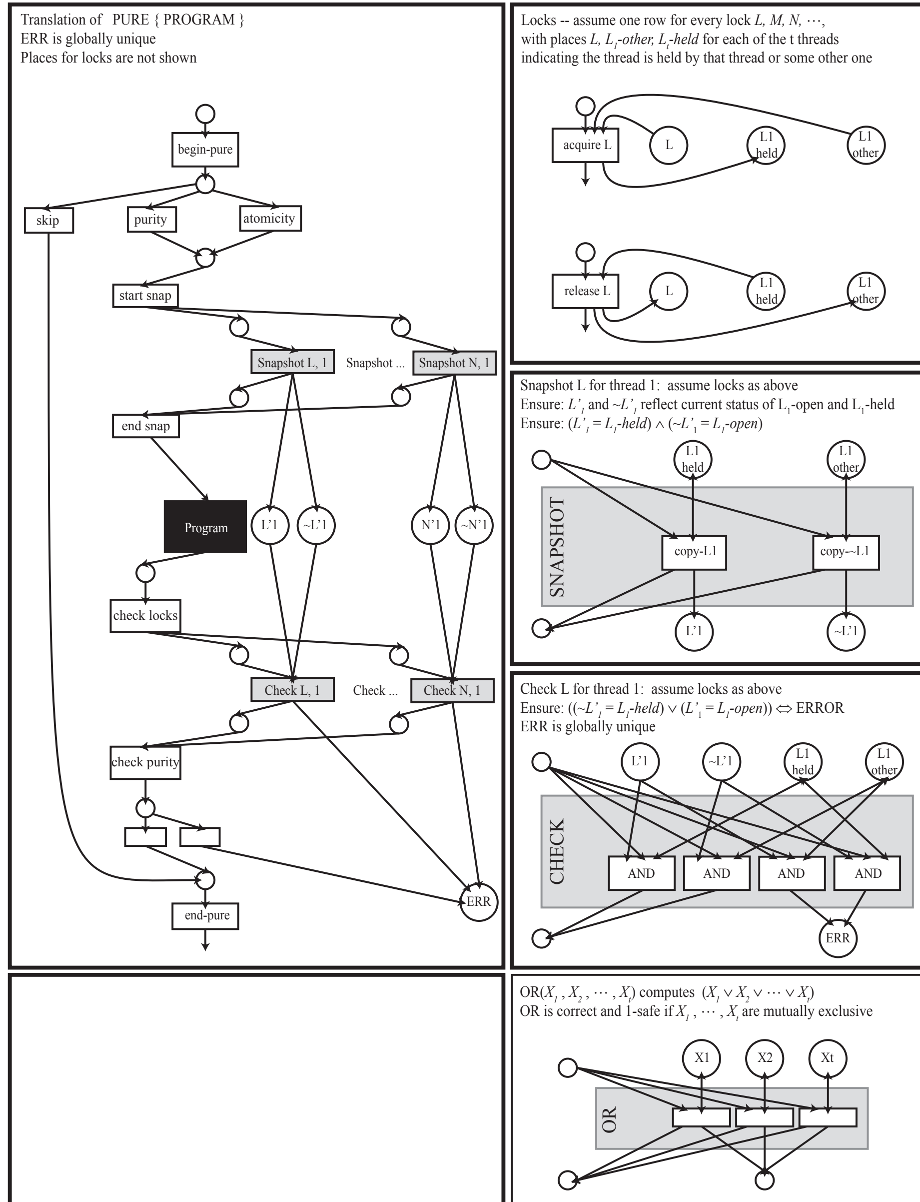

Fig. 8: Widget construction to implement purity checking of locksets and variable accesses. Each lock place L and $Li_{o}ther$ starts off marked. Double-headed arrows indicate a place is both a pre- and post-condition for a transition.

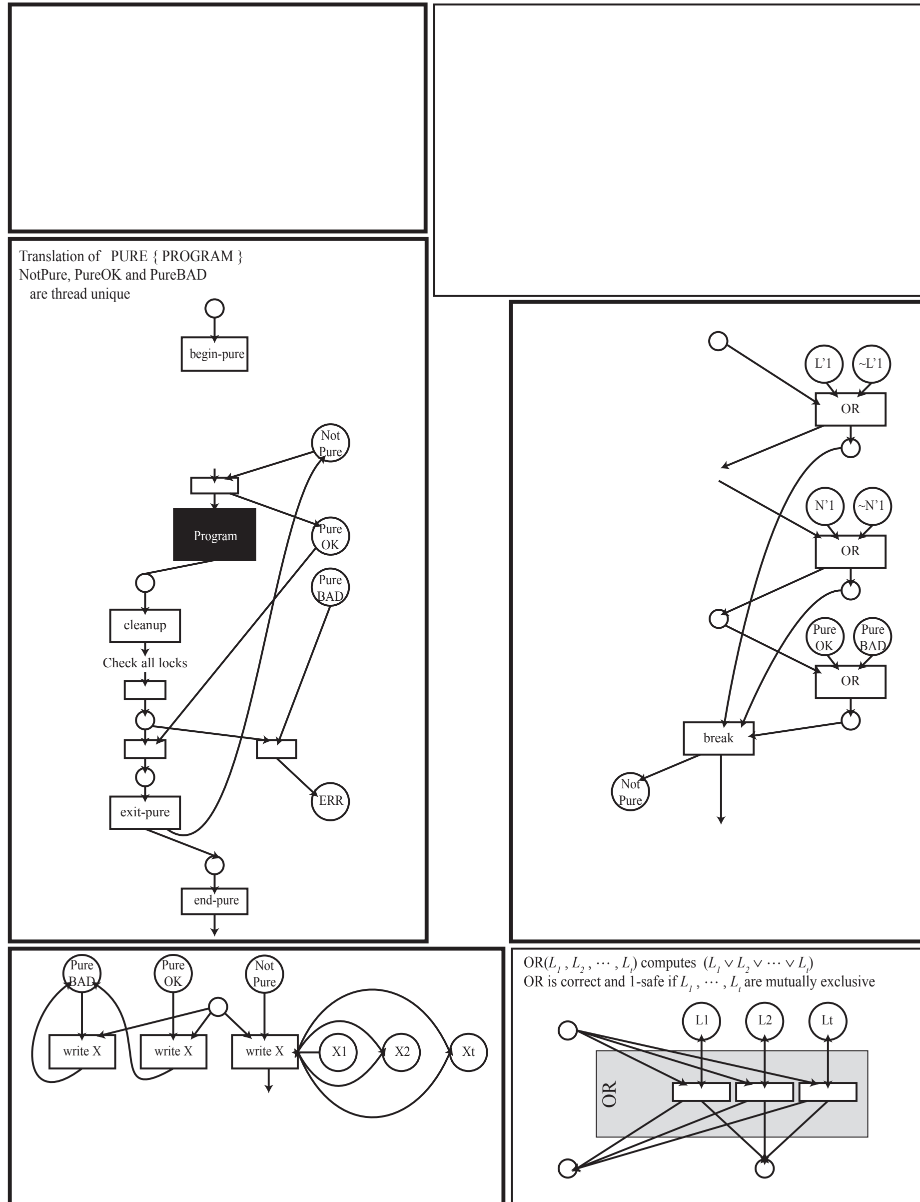

Fig. 9: Widget construction to implement break statements. Writing to variables has changed, as has the bookkeeping needed when entering a pure block. The initial marking for break statements is empty; all markings come from the rest of the net.

that thread. Acquiring a lock moves a mark from L_i-*other* to L_i-*held*; releasing a lock returns it to L_i-*other*.

Translating `pure` statements requires checking that no impure writes to stable variables have occurred, and that the final lockset is equal to the initial lockset. We delegate the responsibility of checking for impure writes to the coloring rules for variable accesses (the informal description given above), leaving us only with the locksets. To track the locksets held at the beginning and end of a pure statement, we need a fairly complicated widget in the Petri net. Essentially, we take a "snapshot" of the lock state of the entire program, copying L_i-*held* and L_i-*other* to two temporary places L_i' and $\neg L_i'$; we do this for every lock L. Note that exactly one of $\neg L_i'$ and L_i' is marked whenever we are within a pure statement (outside such statements, both places are unmarked). At the end of the pure block, we check the currently held locks against the snapshot. If they agree, then we exit the pure statement via the `end-pure` transition; if they disagree, we can reach the ERROR transition and place above. Again, this transition is reachable if and only if the supposedly pure block leaves behind some modified locks, and is therefore not pure.

We are guaranteed that our snapshots accurately reflect the current state of the locks in the program with respect to the current thread (since snapshots do not exist in the source program and are artifacts in our translation, we must take care to prevent any inconsistencies where one thread operates on a lock while another thread is taking a snapshot), since the places L_i-*held*, L_i-*other*, L_i' and $\neg L_i'$ are all thread-local, and thread i cannot be manipulating locks while we are checking its locksets.

Finally, to model the intuitive meaning that a pure statement can be skipped altogether, an extra place is inserted at the beginning of each translated pure statement that chooses precisely one path through the pure statement: either ignoring it completely (via a transition directly to the end of the pure statement), or traversing the body of the statement and the checks inserted along the way.

The net effect of this translation is to instrument every pure statement such that an error location is reachable iff the statement violates the notion of purity defined for Cap.

Break statements serve two purposes: they escape from infinite `loop`s, and they permit impure computation just before exiting a `pure` statement. To achieve both of these, `break` statements must be enclosed by `block` statements, which provide a jump target for the `break`. To finesse the edge case in the translation (where a `break` statement is encountered with no surrounding `block`), we set the initial jump target to ERROR; equivalently, we can syntactically dismiss any such broken programs.

Break statements have no special responsibilities when exiting a `loop`, however, they must behave properly when exiting a `pure` block. There are two conditions on *normal* exits from a `pure` block: no variable writes, and no changed locks. We use a snapshot to handle the latter, and coloring rules for the former. Accordingly, the translation of `break` statements simply skips over the snapshot checking, and resets the color of the mark. Since the `pure` statement might be

contained within a `loop`, we must also reset the state of the snapshots so that they are ready for the next iteration of the loop; this is achieved by taking the OR of all L'_i and $\neg L'_i$ pairs as input, which clears those marks. Finally, control jumps to the place following EXIT-PURE, which then flows out through END-PURE to the rest of the program.

One consequence of this translation is that pure blocks cannot be analyzed if they are syntactically nested within a given thread — this is our reason for the earlier restriction.

5 Checking causal atomicity with purity

Again, we proceed in stages:

1. We define a notion of *pure-causal atomicity.*
2. We define a notion of *causal purity.*
3. We show for Pure-SML that atomicities strictly stronger than A put restrictions on what operations can be performed.
4. We show the race detector is still sufficient.
5. We use the previous two results to show that sequencing two pure-causally atomic statements, whose combination effect-checks as pure and atomic, is itself pure-causally atomic.
6. We use this to show that our translation is correct, that is, for any program which effect-checks, all atomic blocks will be pure-causally atomic.

5.1 Defining pure-causal atomicity

We need an analogue of the definition in Lemma 1 that includes purity. For convenience, we restate the definition here:

Definition 9. *A code block $B =$ `atomic` S in program P is* causally atomic *if and only if* TRANS(P) *does not have a trace tr for which the following condition holds:*

$$\exists e_1^T \in_\lambda \text{START}(S), e_2^T, f^{T'} \in \mathcal{E}.T \neq T' \wedge \\ (e_1 \preceq f \preceq e_2) \wedge (\nexists e^T \in_\lambda \text{END}(S).e_1 \preceq e \preceq e_2)$$

To incorporate purity, we note that the semantics of a `pure` block on successful normal exit is the same as `skip`, and therefore *any causal conflicts should not count.* Moreover, unless a normally-terminating pure block is modelled as `skip`, our results about pure-causal atomicity are invalidated for that trace. Therefore we add a second condition on the trace, that all pure blocks not take the t_{purity} transition, and if they take $t_{atomicity}$, that they not terminate normally. Formally, our new definition is:

$$
\begin{aligned}
&OR(p_{in}, X; p_{out}, p_{ans}; tid) = (T, F) \text{ where}\\
&\quad T = \{t_x \mid x \in X\} X \text{ is any set of places in the Petri net}\\
&\quad F = \{(p_{in}, t_x), (x, t_x), (t_x, p_{out}), (t_x, p_{ans}) \mid x \in X\}\\
&SNAPSHOT(p_{in}, l; p_{out}, p_{l,copy}, p_{\bar{l},copy}; tid) = (T, F) \text{ where}\\
&\quad T = \{t_{l,tid}, t_{\bar{l},tid} \text{ fresh}(tid)\}\\
&\quad F = \{(p_{in}, t_{l,tid}), (l_{tid,held}, t_{l,tid}), (t_{l,tid}, p_{out}), (t_{l,tid}, p_{l,copy}), (t_{l,tid}, l_{tid,held})),\\
&\qquad (p_{in}, t_{\bar{l},tid}), (l_{tid,other}, t_{\bar{l},tid}), (t_{\bar{l},tid}, p_{out}), (t_{\bar{l},tid}, p_{\bar{l},copy}), (t_{\bar{l},tid}, l_{tid,other})\}\\
&CHECK(p_{in}, p_{l,copy}, p_{\bar{l},copy}, l; p_{out}, p_{ERR}; tid) = (T, F) \text{ where}\\
&\quad T = \{t_{hh}, t_{uu}, t_{uh}, t_{hu} \text{ fresh}(tid)\}\\
&\quad F = \{(p_{in}, t_{hh}), (p_{l,copy}, t_{hh}), (l_{tid,held}, t_{hh}), (t_{hh}, p_{out}), (t_{hh}, l_{tid,held})\} \cup\\
&\qquad \{(p_{in}, t_{uu}), (p_{\bar{l},copy}, t_{uu}), (l_{tid,other}, t_{uu}), (t_{uu}, p_{out}, (t_{uu}, l_{tid,other}))\} \cup\\
&\qquad \{(p_{in}, t_{hu}), (p_{l,copy}, t_{hu}), (l_{tid,other}, t_{hu}), (t_{hu}, p_{ERR}), (t_{hu}, l_{tid,other}\} \cup\\
&\qquad \{(p_{in}, t_{uh}), (p_{\bar{l},copy}, t_{uh}), (l_{tid,held}, t_{uh}), (t_{uh}, p_{ERR}), (t_{uh}, l_{tid,held})\}\\
&WIPE(p_{in}; p_{out}; tid; L) = (P, T, F) \text{ where}\\
&\quad P = \{p_{in,l}, p_{out,l}, p_{ans,l} \mid l \in L\}\\
&\quad (T_l, F_l) = OR(p_{in,l}, \{snap_l, snap_{\bar{l}}\}; p_{out,l}, p_{ans,l}; tid) \text{ for each } l \in L\\
&\quad T = \{t_{wipe}, t_{break} \text{ fresh}(tid)\} \cup \bigcup_{l \in L} T_l\\
&\quad F = \{(p_{in}, t_{wipe}), (t_{break}, p_{out})\} \cup\\
&\qquad \{(t_{wipe}, p_{in,l}), (p_{out,l}, t_{break}), (p_{ans,l}, t_{break}) \mid l \in L\} \cup \bigcup_{l \in L} F_l
\end{aligned}
$$

$$\text{TRANS}_s : (Stmt, V, L, tid) \to ((P, T, F), p_{in} \in P, out \subset T, break \subset T)$$

$$
\begin{aligned}
&\text{TRANS}_s(\texttt{block } s, V, L, tid) = ((P, T, F), p_{in}, \{t_{out}\} \cup out_s, \emptyset) \text{ where}\\
&\quad ((P_s, T_s, F_s), p_{in}, out_s, break_s) = \text{TRANS}_s(s, V, L, tid)\\
&\quad P = P_s \cup \{p_{break} \text{ fresh}(tid)\}\\
&\quad T = T_s \cup \{t_{out} \text{ fresh}(tid)\}\\
&\quad F = F_s \cup \{(p_{break}, t_{out})\} \cup \{(t, p_{break}) \mid t \in break_s\}\\
&\text{TRANS}_s(\texttt{acquire } l, V, L, tid) = ((P, T, F), p_{in}, \{t_{acq\ l}\}) \text{ where}\\
&\quad P = \{p_{in} \text{ fresh}(tid), l_{open} \in L, l_{tid} \in L, l_{tid,held} \in L, l_{tid,other} \in L\}\\
&\quad T = \{t_{acq\ l} \text{ fresh}(tid)\}\\
&\quad F = \{(p_{in}, t_{acq\ l}), (l_{other}, t_{acq\ l}), (t_{acq\ l}, l_{tid}), (l_{tid,other}, t_{acq\ l}), (t_{acq\ l}, l_{tid,held})\}\\
&\text{TRANS}_s(\texttt{release } l, V, L, tid) = ((P, T, F), p_{in}, \{t_{rel\ l}\}) \text{ where}\\
&\quad P = \{p_{in} \text{ fresh}(tid), l_{open} \in L, l_{tid} \in L, l_{tid,held} \in L, l_{tid,other} \in L\}\\
&\quad T = \{t_{rel\ l} \text{ fresh}(tid)\}\\
&\quad F = \{(p_{in}, t_{rel\ l}), (l_{tid}, t_{rel\ l}), (t_{rel\ l}, l_{other}), (l_{tid,held}, t_{acq\ l}), (t_{acq\ l}, l_{tid,other})\}
\end{aligned}
$$

Fig. 10: Auxiliary widgets used to construct TRANS(`pure` s) and TRANS_s(`break`).

$\text{TRANS}_s : (Stmt, V, L, tid) \to ((P, T, F), p_{in} \in P, out \subset T, break \subset T)$

$\text{TRANS}_s(\texttt{pure}\ s, V, L, tid) = ((P, T, F), p_{in}, \{t_{end_pure}\}, break)$ where

$((P_s, T_s, F_s), p_s, out_s, breaks_s) = \text{TRANS}_s(s, V, L, tid)$

If $breaks_s \neq \emptyset$:

p_{wipe}, p_{break} fresh(tid)

$(P', T', F') = WIPE(p_{wipe}; p_{break}; tid)$

$P_{wipe} = \{p_{wipe}, p_{break}\} \cup P'$

$T_{wipe} = \{t_{break} \text{ fresh}(tid)\} \cup T'$

$F_{wipe} = \{(t, p_{wipe}) \mid t \in breaks_s\} \cup F' \cup \{(p_{break}, t_{break})\}$

$break = \{t_{break}\}$

Else

$break = breaks_s$

$P_{wipe} = T_{wipe} = F_{wipe} = \emptyset$

$P = P_s \cup P_{wipe} \cup \{Lock_Snap\} \cup \{p_{in}, p_1, p_2, p_3, p_4, p_5 \text{ fresh}(tid)\} \cup$
$\{snap_{in,l}, snap_{out,l}, snap_l, snap_{\bar{l}}, check_{in,l}, check_{out,l} \text{ fresh}(tid) \mid l \in L\}$

$(T_{snap,l}, F_{snap,l}) = SNAPSHOT(snap_{in,l}, l; snap_{out,l}, snap_l, snap_{\bar{l}}; tid)$ and

$(T_{check,l}, F_{check,l}) = CHECK(check_{in,l}, snap_l, snap_{\bar{l}}, l; check_{out,l}, p_{ERR}; tid)$

for each $l \in L$

$$T = \{t_{begin_pure}, t_{skip_all}, t_{snap_start}, t_{snap_end}, t_{check_locks}, t_{check_purity}, t_{pureOK}, t_{pureBad}, t_{exit_pure}, t_{end_pure} \text{ fresh}(tid)\} \cup F_{wipe} \cup \bigcup_{l \in L} T_{snap,l} \cup T_{check,l}$$

$$\begin{aligned} F = {} & \{(p_{in}, t_{begin_pure}), (t_{begin_pure}, p_1), (p_1, t_{skip_all}), (t_{skip_all}, p_5)\} \cup \\ & \{(p_1, t_{snap_start}), (Lock_Snap, t_{snap_start}), (t_{snap_end}, Lock_Snap)\} \cup \\ & \{(t_{snap_start}, snap_{in,l}), (snap_{out,l}, t_{snap_end}) \mid l \in L\} \cup \\ & \{(t_{snap_end}, p_s)\} \cup \{(t, p_2) \mid t \in out_s\} \cup \\ & \{(p_2, t_{check_locks}), (Lock_Snap, t_{check_locks})\} \cup \\ & \{(t_{check_locks}, check_{in,l}), (check_{out,l}, t_{check_purity}) \mid l \in L\} \cup \\ & \{t_{check_purity}, p_3), (p_3, t_{pureOK}), (p_3, t_{pureBad}), (t_{pureBad}, p_{ERR}), \\ & (t_{pureOK}, p_4), (p_4, t_{exit_pure}), (t_{exit_pure}, Lock_Snap), \\ & (t_{exit_pure}, p_5), (p_5, t_{end_pure})\} \cup F_{wipe} \cup \bigcup_{l \in L} F_{snap,l} \cup F_{check,l} \end{aligned}$$

$\text{TRANS}_s(\texttt{break}, V, L, tid) = ((P, T, \{(p_{in}, t_{break})\}), p_{in}, \emptyset, \{t_{break}\})$ where

$P = \{p_{in} \text{ fresh}(tid)\} \cup \{p_{in,l}, p_{out,l}, p_{ans,l} \mid l \in L\}$

$T = \{t_{break} \text{ fresh}(tid)\}$

Fig. 11: Pseudocode of TRANS($Prog$), TRANS(Exp) and TRANS($Stmt$). All "fresh(tid)" places and transitions are meant to be new and marked as part of thread tid, even if their names have appeared before in P or T.

Definition 10. *A trace through the translation of a program P is* non-pure *if and only if the following condition holds:*

$$\forall e \in \mathcal{E}.(\exists C.e \in \text{CURRENT}(\texttt{pure } C, e) \implies \nexists e_p^T \in \text{CURRENT}(\texttt{pure } C, e).\lambda(e_p) = t_{purity} \wedge (\exists e_a^t \in \text{CURRENT}(\texttt{pure } C, e).\lambda(e_a) = t_{atomicity} \implies \exists e_b^t \in \text{CURRENT}(\texttt{pure } C, e).\lambda(e_b) = t_{break}))$$

A code block $B =$ `atomic` S in program P is pure-causally atomic *if and only if* TRANS(P) *does not have a non-pure trace tr for which the following condition holds:*

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

where T and T' denote distinct threads.

This definition trivially extends the previous one: for programs in SML that have no `pure` blocks, the distinction between all traces and non-pure traces evaporates, so (ignoring the slight change in writing `while` loops) any causally atomic program is also pure-causally atomic.

Lemma 8. *If a code block B is causally atomic, then it is pure-causally atomic.*

Proof. Obvious. □

5.2 Causal purity

As with atomicity, we need a notion of purity defined in terms of behaviors over the Petri net.

Definition 11. *A code block B in program P is* causally pure *if and only if all traces through* TRANS(P) *satisfy the following:*

1. *All acquired locks must be released on normal exit:*

$$\forall e \in_\lambda \text{TRANS}(B).(\nexists b \in \text{CURRENT}(B, e).\lambda(b) = t_{break}) \implies (\lambda(e) = t_{acq\ l} \wedge \nexists e' \in \text{CURRENT}(B, e).e' \preceq e \wedge \lambda(e') = t_{rel\ l}) \implies \exists f \in \text{CURRENT}(B, e).(e \preceq f \wedge \lambda(f) = t_{rel\ l} \wedge \nexists f' \in \text{CURRENT}(B, e).f \preceq f' \wedge \lambda(f') = t_{acq\ l})$$

2. *All released locks must be reacquired on normal exit:*

$$\forall e \in_\lambda \text{TRANS}(B).(\nexists b \in \text{CURRENT}(B, e).\lambda(b) = t_{break}) \implies (\lambda(e) = t_{rel\ l} \wedge \nexists e' \in \text{CURRENT}(B, e).e' \preceq e \wedge \lambda(e') = t_{acq\ l}) \implies \exists f \in \text{CURRENT}(B, e).(e \preceq f \wedge \lambda(f) = t_{acq\ l} \wedge \nexists f' \in \text{CURRENT}(B, e).f \preceq f' \wedge \lambda(f') = t_{rel\ l})$$

3. No variable writes are permitted on normal exit:

$$\forall e \in_\lambda \text{TRANS}(B).(\lambda(e) = t_x \wedge t_x \textit{ is a variable write}) \implies \exists f \in \text{CURRENT}(B, e).e \preceq f \wedge \lambda(f) = t_{break}$$

Note that this definition requires that any impure actions require the pure block to `break` and terminate, and that any infinite execution of the block must therefore not perform any impure actions.

From this definition we can show the following:

Lemma 9. *Assume a program* $P \in$ Pure-Sml *such that* $\vdash P : ok$*. Consider a statement (or expression) s in P, and let* $N = \text{TRANS}(s)$*. If* $\Gamma, X, \bot, \Sigma \vdash s : \bot, \Sigma$*, then N is causally pure.*

Proof. By induction on the structure of s:

- Case Stmt-Skip: $s =$ `skip`. From the typing rule, we have the requested condition. The translation of `skip` is a single transition that does not access variables or locks, so is vacuously causally pure according to the definition.
- Case Stmt-Break: $s =$ `break`. Immediate by the same reasoning as above.
- Case Stmt-Pure: $s =$ `pure` s'. We know that $\vdash P : ok$, and therefore $\Gamma, X, \bot, \Sigma \vdash$ `pure` $s' : \bot, \Sigma$, which requires $\Gamma, X \cap UnstableVar, \bot, \Sigma \vdash s' : \bot, \Sigma$. By induction, we therefore have that TRANS(s') is causally pure. But TRANS(s) \ TRANS(s') involves only "administrative" transitions that do not break, set locks, or access variables, and therefore cannot violate the definition of causal purity. This gives the causal purity of TRANS(s).
- Case Stmt-Assign and Stmt-Assign-Race: $s = x := exp$. We cannot have $\Gamma, X, \bot, \Sigma \vdash s : \bot, \Sigma$ at all, so the lemma is vacuously true.
- Case Stmt-Seq: $s = s_1; s_2$. We have $\Gamma, X, \bot, \Sigma \vdash s_1 : p_1, \Sigma_1$ and $\Gamma, X, p_1, \Sigma_1 \vdash s_2 : p_2, \Sigma_2$. Further, we must have $\Sigma_2 = \Sigma$ and $p_2 = \bot$, or else the lemma is vacuously true; however, we know nothing about Σ_1 and p_1. Thus we can't inductively state anything about s_1 or s_2. Resorting to the definition of causal purity, we have three conditions to check:
  1. All acquired locks must be released on normal exit: Suppose that $\exists e \in_\lambda \text{TRANS}(s).\lambda(e) = t_{acq\ l} \wedge \nexists e' \in \text{CURRENT}(s, e).e' \preceq e \wedge \lambda(e') = t_{rel\ l}$ as stipulated by the definition. By examining the TRANS construction, we know s must contain `acquire` l. Moreover, we know that since e is the first event to acquire or release l, that no other `acquire` or `release` statements precede this one. Therefore we know that $l \notin \Sigma$ at the beginning of s. Since all the rules except for Stmt-Release are monotonically non-decreasing in Σ, it is obvious that if a statement `acquire`s a lock l and terminates normally (i.e. with no `break` statements), it must retain that lock at exit. We know that s terminates normally with Σ locks held, and that $l \notin \Sigma$. Therefore we know s must `release` l, as this is the only statement whose typing rule is not monotonic and can remove l from Σ. (Note: the lock could be acquired and released multiple times, but the final operation must be a release.) Examining TRANS again, we see that s

must contain `release` l on all paths following the `acquire`, or else l would remain in Σ. Therefore $\exists f \in_\lambda$ CURRENT$(s,e).e \preceq f \wedge \lambda(f) = t_{rel\ l}$. Moreover, since `release` must be the final operation on l, we know that for at least one f, $\nexists f' \in$ CURRENT$(s,e).f \preceq f' \wedge \lambda(f') = t_{acq\ l}$, as required.

2. All released locks must be reacquired on normal exit: This is symmetric to the previous case.
3. No variable writes are permitted on normal exit: Suppose that $\exists e \in_\lambda$ TRANS$(s).\lambda(e) = t_x$ and t_x is a variable write. We know from TRANS that s must contain an assignment $x := exp$. From the typing rules, we know that $\Gamma, X, p, \Sigma \vdash x := exp : \top, \Sigma$ for any p. Since all the rules except for Stmt-Break are monotonic, it is obvious that if a statement contains an assignment and terminates normally (i.e. with no `break` statements), it must have purity $\top$. We know that $p_2 = \bot$. Therefore, there must be a `break` statement within s, as this is the only statement whose typing rule is not monotonic and can return the purity of the statement to $\bot$. Moreover, some `break` statement must follow on every path after the assignment, or else the resulting purity will remain at $\top$. Therefore, we must have some event $f \in$ CURRENT$(s,e).e \preceq f \wedge \lambda(f) = t_{break}$, as required.

– Case Stmt-Loop: $s =$ `loop` s'. We have $\Gamma, X, p, \Sigma \vdash s : p', \Sigma$, so by induction we know that TRANS(s') is causally pure. Since TRANS$(s) \setminus$ TRANS(s') involes only adding back-edges from the exit transitions of s' to the entry place of s, and does not construct arcs involving variables or locks, we are done.We have $\Gamma, X, p, \Sigma \vdash s : p', \Sigma$, so by induction we know that TRANS(s') is causally pure. Since TRANS$(s) \setminus$ TRANS(s') involes only adding back-edges from the exit transitions of s' to the entry place of s, and does not construct arcs involving variables or locks, we are done.
– Case Stmt-If: $s =$ `if` $exp\ s_1\ s_2$. Examining the translation of `if` statements, we see that it constructs two transitions t_t and t_f, and recursively constructs TRANS(exp), TRANS(s_1), and TRANS(s_2). Since none of the transitions in an expression write variables, access locks or break, and clearly neither do t_t or t_f, we have that s is causally pure if and only if TRANS(s_1) and TRANS(s_2) are causally pure. Looking at the typing rules, we see that we must have $\Gamma, X, p, \Sigma \vdash s_1 : p_1, \Sigma$ and $\Gamma, X, p, \Sigma \vdash s_2 : p_2, \Sigma$. Further, $p = \bot$ (because s typechecked using $\bot$), and further we know $p_1 \sqcup p_2 = \bot$, hence $p_1 = p_2 = \bot$. Therfore we can apply induction to both statements to prove their causal purity, and are done.
– Case Stmt-Atomic: $s =$ `atomic` s'. Examining the translation of atomic statements, we see that it recursively constructs TRANS(s'), and so is causally pure if and only if s' is casually pure. Looking at its typing rule, we see that s' must typecheck under the same purities and locksets as s itself, and so by induction we are done.
– Case Stmt-Acquire and Stmt-Release: $s =$ `acquire` l or $s =$ `release` l. We cannot have $\Gamma, X, \bot, \Sigma \vdash s : \bot, \Sigma$, so we are vacuously true.

□

It follows that

Lemma 10. *Assume a program* $P \in$ Pure-Sml *such that* $\vdash P : ok$*. For every statement* $s = \texttt{pure}\ s'$ *in* P*,* TRANS(s) *is causally pure.*

Proof. By the typing rules for `pure` blocks, we know $\Gamma, X \cap UnstableVar, \bot, \Sigma \vdash s' : \bot, \Sigma$. By the lemma above, this means TRANS(s') is causally pure. Of the transitions in TRANS(s) \ TRANS(s'), none are accesses to variables or locks, so the conditions for causal purity are vacuously satisfied. Therefore TRANS(s) is causally pure as well. □

5.3 Enumerating the operations of movers

Lemma 11. *Consider a program* $P \in$ Pure-Sml *such that* $\vdash P : ok$*, and a statement* s *in* P *such that* $\Gamma \vdash s : a \uparrow b$ *and* $a, b \sqsubset A$*, and* $\Gamma, X, p, \Sigma \vdash s : p', \Sigma'$*. For an arbitrary trace, if there exists an event* $e^T \in_\lambda$ TRANS(s) *and there exists another event* $f^{T'}$ *where* $T \neq T'$ *such that that* $e \prec f$ *or* $f \prec e$*, then:*

1. *If* $a \sqsubseteq R$*, then* $\lambda(e) = t_x$ *and* $\Gamma(x) \in \Sigma$ *or* $\lambda(e) = t_{acq\ l}$*; that is,* e *must be a race-free variable access or a lock-aquire.*
2. *If* $a \sqsubseteq L$*, then* $\lambda(e) = t_x$ *and* $\Gamma(x) \in \Sigma$ *or* $\lambda(e) = t_{rel\ l}$*; that is,* e *must be a race-free variable access or a lock-release.*

Proof. The proof is almost identical to that of Lemma 2, but we add new cases for the new constructs in Pure-Sml:

– Case Stmt-Pure: $s = \texttt{pure}\ s'$. Suppose $e \in_\lambda$ TRANS(s'); by induction we are done. Examining TRANS(s) \ TRANS(s'), we see that all transitions are connected solely to thread-local places, and so are dependent only other transitions in the current thread, so we are vacuously true.
– Case Stmt-Break: $s = \texttt{break}$. TRANS(s) contains no transitions that interact with anything in other threads, so we are vacuously true.
– Case Stmt-Block: $s = \texttt{block}\ s'$. Suppose $e \in_\lambda$ TRANS(s'); by induction we are done. Examining TRANS(s)\TRANS(s'), we see that only one transition is constructed and it does not interact with any transitions in other threads, so we are vacuously true.
– Case Stmt-Loop: $s = \texttt{loop}\ s'$. Suppose $e \in_\lambda$ TRANS(s'); by induction we are done. Examining TRANS(s)\TRANS(s'), we see that only one transition is constructed and it does not interact with any transitions in other threads, so we are vacuously true. □

5.4 Sufficiency of our race detector

We need to show that the race analysis we encoded above is sufficient, and we do this in two steps:

Lemma 12. *Consider a progam $P \in$ Pure-SML such that $\vdash P : ok$, and its translation* TRANS(P)*. Consider a statement s in thread T such that $\Gamma \vdash s : a \uparrow b$ and $\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''$ occurs within the derivation of $\vdash P : ok$. For an arbitrary trace, consider an arbitrary event $e_e^T \in_\lambda \mathrm{END}(s)$. Suppose that*

$$\begin{aligned}\forall e_s^T \in_\lambda \mathrm{START}(s), e_s \in \mathrm{CURRENT}(s, e_e). \forall l \in \Sigma. \\ \exists acq_l^T \in \mathcal{E}. acq_l \preceq e_s \wedge \lambda(acq_l) = t_{acq\ l} \wedge \\ \nexists rel_l^T . acq_l \preceq rel_l \preceq e_s \wedge \lambda(rel_l) = t_{rel\ l}^T\end{aligned}$$

Then if $\lambda(e_e) \neq t_{break}$, then

$$\begin{aligned}\forall f^T \succ e_e. \forall l \in \Sigma'. \exists acq_l^T . \\ acq_l \preceq f \wedge \lambda(acq_l) = t_{acq\ l}^T \wedge \\ \nexists rel_l^T . acq_l \preceq rel_l \preceq f \wedge \lambda(rel_l) = t_{rel\ l}^T\end{aligned}$$

and if $\lambda(e_e) = t_{break}$, then

$$\begin{aligned}\forall f^T \succ e_e. \forall l \in \Sigma''. \exists acq_l^T . \\ acq_l \preceq f \wedge \lambda(acq_l) = t_{acq\ l}^T \wedge \\ \nexists rel_l^T . acq_l \preceq rel_l \preceq f \wedge \lambda(rel_l) = t_{rel\ l}^T\end{aligned}$$

In other words, if a set of locks Σ is held before a statement s executes in the Petri net, and s terminates normally yielding a new set of locks Σ', then that set of locks truly is held after s executes, and if it terminates abruptly yielding a new set of locks Σ'', then that set of locks truly is held after s executes.

Proof. By induction on $\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''$.

- Case Stmt-Atomic, Stmt-Assign, Stmt-Assign-Race, Stmt-Skip: None of these operations produce any events that acquire or release locks, except inductively on their substatements (if any). See the sequence case.
- Case Stmt-Seq: $s = (s_1; s_2)$. We assume $\Gamma, \Sigma \vdash s : \Sigma_s' \uparrow \Sigma_s''$. By assumption we know that e_e exists, and that $\mathrm{END}(s) = \mathrm{END}(s_2) \cup \{t_{break} \in \mathrm{TRANS}(s_1)\}$. If $e_e \in_\lambda \mathrm{TRANS}(s_1)$, then we know that s_1 terminated abruptly. By the typing rules, we must have $\Gamma, \Sigma \vdash s_1 : \Sigma_1' \uparrow \Sigma_1''$, and $\Gamma, \Sigma_1' \vdash s_2 : \Sigma_2' \uparrow \Sigma_2''$, where $\Sigma_s'' = \Sigma_1'' \cap \Sigma_2''$, and Σ_1' is unconstrained. By induction, if s_1 terminated abruptly it truly yields a lockset Σ_1''; when s terminates abruptly in the same way, $\Sigma_1'' \cap \Sigma_2''$ clearly is held.
  Otherwise, we know $e_e \in_\lambda \mathrm{END}(s_2)$, which means that s_2 executes. By the construction of TRANS(s), we know that s_1 must have terminated in order for s_2 to execute; therefore, $\exists e_1 \in \mathrm{LAST}(s_1, e_e). e_1 \in_\lambda \mathrm{END}(s_1) \wedge \lambda(e_1) \neq t_{break}$, which in turn means that s_1 terminated normally. Using the same notation as above, now we know for certain Σ_1', and Σ_1'' is unconstrained. By induction, we can therefore say that if some Σ_1' is held just prior to this execution of s_2, then Σ_2' will be held after it when terminating normally, and Σ_2'' will be held after it when terminating abruptly. This in turn implies the desired result.

- Case Stmt-If: Suppose $s = \texttt{if}\ e\ s_1\ s_2$. Then $TRANS(s)$ contains two transitions t_t and t_f, as well as the recursive constructions on the substatements. Trivially, t_t and t_f are not $t_{rel\ l}$. Also trivially, none of the transitions in TRANS(e) are $t_{rel\ l}$; moreover, e always terminates normally. This leaves only the substatements s_1 and s_2. We know that $\text{END}(s) = \text{END}(s_1) \cup \text{END}(s_2)$.
  Suppose $e_e \in_\lambda \text{END}(s_1)$ (the other case with s_2 is symmetric). We have $\Gamma, \Sigma \vdash s_1 : \Sigma \uparrow \Sigma'$ from the typing derivation. Let e_s be as given in the assumption. For all $e_1^T \in_\lambda \text{START}(s_1)$ such that $e_s \preceq e_1 \preceq e_e$, we know that Σ is held before e_1 by the above reasoning. By induction, we know that Σ is held when s_1 terminates normally, and Σ' is held when it terminates abruptly; since $\text{END}(s_1) \subseteq \text{END}(s)$ this implies the desired result.
- Case Stmt-Loop: We have $s = \texttt{loop}\ s'$. Examining TRANS(s) we see that $\text{END}(s) = \{t_{break} \in \text{TRANS}(s')\}$, and that s never terminates normally, and that $\text{START}(s) = \{t_{head}\}$. Now, we know $\exists e_1 \in_\lambda \text{START}(s').e_s \preceq e_1 \preceq e_e$. By construction, either $e_s \prec e_1$, or $\exists e_2 \in_\lambda \text{END}(s').\lambda(e_2) \neq t_{break} \wedge e_2 \prec e_1$; this latter case corresponds to iterating the loop. Since we know that every evnet in a trace has a finite set of predecessors, we know that the loop must have iterated only a finite number of times; we now use induction over the number of iterations. If $e_s \prec e_1$ we are trivially done, as the loop has yet to execute even onces, and e_s does not modify Σ. Otherwise, the loop has iterated $n-1$ times and we are checking the nth iteration. From the typing rules we are guaranteed $\Gamma, \Sigma \vdash s' : \Sigma \uparrow \Sigma'$, and we know the first $n-1$ iterations terminated normally. Therefore we effectively have a sequence $s'; s'; \cdots; s'$, and the same logic shows that if Σ was held before the last iteration of the loop, then Σ will be held if the last iteration terminates normally, and Σ' will be held if it terminates abruptly. This shows that if the loop iterates n times, the correct locksets will truly be held. Finally, we know that the last iteration of the loop terminates abruptly (because END(s) only contains t_{break} transitions), and we know we are left with Σ' held after abrupt termination. The lockset for normal termination, Σ'', is completely unconstrained.
- Case Stmt-Acquire: We have $\Gamma, \Sigma \vdash \texttt{acquire}(l) : \Sigma \cup \{\, l \,\} \uparrow \Sigma''$. There is exactly one transition in TRANS(s), so $\text{START}(s) = \text{END}(s) = \{t^T_{acq\ l}\}$; moreover, $e_s = e_e$. Σ'' is completely unconstrained. For a lock $l' \in \Sigma'$:
  - If $l' = l$, then we trivially satisfy the condition above: we let $acq_l^T = e_s^T$.
  - If $l' \neq l$, then we know from the premise and from the observation that $e_s = e_e$ that

    $$\exists acq_{l'}^T.acq_{l'} \preceq e_e \wedge \lambda(acq_{l'}) = t^T_{acq\ l'} \wedge \\ \nexists rel_{l'}^T.acq_{l'} \preceq rel_{l'} \preceq e_e \wedge \lambda(rel_{l'}) = t^T_{rel\ l'}$$

    Since s doesn't touch l', we must have that this holds for all $f^T \succ e_e^T$, as desired.
- Case Stmt-Release: We have $\Gamma, \Sigma \vdash \texttt{release}(l) : \Sigma \setminus \{\, l \,\} \uparrow \Sigma''$. By the same argument as above, since s doesn't touch l' for $l' \neq l$, we satisfy the condition.

- Case Stmt-Block: $s = \texttt{block}\ s'$. Examining TRANS(s), we see that it constructs one new transition that does not modify locksets, that START(s) = START(s'), and that END(s) = END(s') $\setminus \{t_{break} \in$ TRANS(s')$\}$ — blocks cannot terminate abruptly. Inductively, we know that $\Gamma, \Sigma \vdash s' : \Sigma' \uparrow \Sigma''$. Therefore, no matter how s' terminates, we are guaranteed that $\Sigma' \cap \Sigma''$ is held, exactly as required. The lockset for abrupt termination of s is completely unconstrained.
- Case Stmt-Break: $s = \texttt{break}$. Examining TRANS(s) we see that it only ever terminates abruptly, and does not modify any locks in the process. So we are guaranteed that Σ will hold on abrupt termination; the lockset for normal termination is completely unconstrained.
- Case Stmt-Pure: $s = \texttt{pure}\ s'$. Examining TRANS(s), we see that it constructs several transitions, none of which manipulate locks (none of them are t_{acq} or t_{rel}, and none of them touch the shared lock places), and recursively constructs TRANS(s'). By induction we have $\Gamma, \Sigma \vdash s' : \Sigma' \uparrow \Sigma''$, and we are done.

□

Lemma 13. *Consider a program $P \in$ Sml such that $\vdash P : ok$. Then for every statement s such that $\Gamma, \Sigma \vdash s : \Sigma_1 \uparrow \Sigma_2$ appears in the derivation of $\vdash P : ok$, if*

$$\begin{aligned}\forall e^T \in_\lambda \text{START}(s).\forall l \in \Sigma.\exists acq_l^T.&\\ &acq_l \preceq e \wedge \lambda(acq_l) = t_{acq\ l}^T \wedge\\ &\nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t_{rel\ l}^T\end{aligned}$$

then for every statement (or expression) s' that appears within s, such that $\Gamma, \Sigma' \vdash s' : \Sigma_1' \uparrow \Sigma_2'$ appears in the derivation of $\vdash P : ok$ we have the same property:

$$\begin{aligned}\forall e^T \in_\lambda \text{START}(s').\forall l \in \Sigma'.\exists acq_l^T.&\\ &acq_l \preceq e \wedge \lambda(acq_l) = t_{acq\ l}^T \wedge\\ &\nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t_{rel\ l}^T\end{aligned}$$

Proof. Expressions do not change locksets, so if a lockset Σ holds at the beginning of an expression *exp*, it holds at the beginning of every subexpression of *exp*. Therefore we need only examine statements. By induction on the structure of s:

- Case $s = \texttt{skip}$: vacuously true.
- Case $s = \texttt{break}$: vacuously true.
- Case $s = x := exp$: Looking at TRANS(s), we see that START(s) = START(exp) and $\Sigma' = \Sigma$, so by induction we are done.
- Case $s = \texttt{if}\ c\ s_1\ s_2$: Looking at TRANS($s$), we see that START($s$) = START($c$), so Σ holds at the beginning of c. Since the transitions t_t and t_f do not modify locks, we know that Σ holds at the beginning of s_1 and s_2 as well; by induction, we are done.

– Case $s = \texttt{acquire}\ l$ or $s = \texttt{release}\ l$: vacuously true.
– Case $s = \texttt{atomic}\ s'$: Since START(s) = START(s'), by induction we are done.
– Case $s = \texttt{block}\ s'$: Since START(s) = START(s'), by induction we are done.
– Case $s = \texttt{pure}\ s'$: Looking at TRANS(s), we see that we construct several transitions, none of which manipulate locks. So $\forall e \in_\lambda \text{START}(s).\forall e' \in_\lambda \text{START}(s').\forall f^T.e \preceq f \preceq e'.\Sigma$ is held. Therefore Σ is held at the beginning of s', and by induction we are done.
– Case $s = (s_1; s_2)$: Since START(s) = START(s_1), by induction we are done with s_1. By the typing rules, we must have that $\Gamma, \Sigma \vdash s_1 : \Sigma_1' \uparrow \Sigma_1''$ and $\Gamma, \Sigma_1' \vdash s_2 : \Sigma_2' \uparrow \Sigma_2''$. If an event e_2 exists in START(s_2), then s_1 must have terminated normally. We can pick any $e_1 \in \text{LAST}(s_1, e_2).\lambda(e_1) \neq t_{break}$, which implies $e_1 \prec e_2$, and by Lemma 12, we know that Σ_1 is held just after e_1, which is to say, just before e_2, so by induction we are done with s_2.
– Case $s = \texttt{loop}\ s'$: The transition t_{head} obviously does not modify any locks, so we need a second level of induction here, since the body s' can execute multiple times. We must show that if the loop has run for a finite number of iterations (n), then on the $n+1$ execution of c, the locks Σ are held just prior to the start of c. It is clear that the set $\{e \in_\lambda \text{START}(s')\} = \{e_0, e_1, \ldots\}$ is the totally ordered set of events corresponding to each iteration of s': event e_n occurs when the loop has executed n times. Let $f_n \prec e_n$ be the event just prior to e_n in the same thread. By induction on n:
  • Case $n = 0$: The condition has not executed at all, therefore $\lambda(f_0) = t_{head}$, and so the same locks Σ are held before e_0 as were held at the start of the loop.
  • Case $n > 0$: Suppose the loop has executed n times, and we're about to execute e_n. Then $f_n \in_\lambda \text{END}(s')$ and $f_n \prec e_n$. By induction, the locks Σ are held before e_{n-1}. Then by Lemma 12, some lockset Σ' is held after f_n, and by the Stmt-Loop rule, we know that $\Sigma' = \Sigma$. Therefore, the lockset Σ is held just prior to e_n as well, as required. Moreover, we know that s' terminates normally, because we supposed event e_n exists. Therefore at every iteration, the lockset Σ is held just prior to its execution.

□

The key corollary to this is

Lemma 14. *Consider a program $P \in$ Pure-Sml such that $\vdash P : ok$. Then for every statement s such that $\Gamma, \Sigma \vdash s : \Sigma' \uparrow \Sigma''$ appears in the derivation of $\vdash P : ok$, we know that*

$$\begin{aligned} \forall e^T \in_\lambda \text{START}(s).\forall l \in \Sigma.\exists acq_l^T.\ & \\ & acq_l \preceq e \wedge \lambda(acq_l) = t_{acq\ l}^T \wedge \\ & \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t_{rel\ l}^T \end{aligned}$$

Proof. We know $P = s_1||s_2||\dots||s_n$. For every statement s_i, we know $\Gamma, \emptyset \vdash s_i : \Sigma' \uparrow \Sigma''$, and we know all threads do start with no locks held in the Petri net, we can apply the above lemma. □

We can therefore conclude

Lemma 15. *Our race detector is sufficient.*

Proof. Consider two accesses to variable x in different threads that are determined to be race-free, as required by the definition of sufficient race detectors; let s and t be the primitive statements or expressions in which these accesses occur, and T and T' be their threads respectively. By Lemma 14, we know that

$$\begin{aligned} &\forall e^T \in_\lambda \text{START}(s).\forall l \in \Sigma.\exists acq_l^T. \\ &\qquad acq_l \preceq e \wedge \lambda(acq_l) = t^T_{acq\ l} \wedge \\ &\qquad \nexists rel_l^T.acq_l \preceq rel_l \preceq e \wedge \lambda(rel_l) = t^T_{rel\ l} \end{aligned}$$

(and similarly for statement t) and in particular, since the accesses are race-free,

$$\begin{aligned} &\forall e^T \in_\lambda \text{START}(s).\exists acq^T_{\Gamma(x)}. \\ &\qquad acq_{\Gamma(x)} \preceq e \wedge \lambda(acq_{\Gamma(x)}) = t^T_{acq\ \Gamma(x)} \wedge \\ &\qquad \nexists rel^T_{\Gamma(x)}.acq_{\Gamma(x)} \preceq rel_{\Gamma(x)} \preceq e \wedge \lambda(rel_{\Gamma(x)}) = t^T_{rel\ \Gamma(x)} \end{aligned}$$

We can conclude that $acq^T_{\Gamma(x)} \preceq e^T_s$, for the specific event $\lambda(e^T_s) = t^T_x$ that accesses x in statement s, and similarly $acq^{T'}_{\Gamma(x)} \preceq e^{T'}_t$ for the event $\lambda(e^{T'}_t) = t^{T'}_x$ that accesses x in statement t. Assume that $e^T_s \preceq e^{T'}_t$ — they must be ordered since they access the same variable (the opposite ordering is symmetric). Therefore $acq^T_{\Gamma(x)} \preceq e^T_s \preceq e^{T'}_t$. We know that $acq^T_{\Gamma(x)} \preceq acq^{T'}_{\Gamma(x)}$ or vice versa, since they access the same lock. We proceed by cases to determine when $acq^{T'}_{\Gamma(x)}$ can occur (a symmetric argument holds when thread T' starts first):

1. $acq^{T'}_{\Gamma(x)} \preceq acq^T_{\Gamma(x)} \preceq e^T_s \preceq e^{T'}_t$. This is a contradiction: from the line above we know that the lock $\Gamma(x)$ is continually held from $acq^{T'}_{\Gamma(x)}$ until $e^{T'}_t$ (since no release event happens between these two events), therefore the place $\Gamma(x)_{open}$ is unmarked, so therefore $acq^T_{\Gamma(x)}$ cannot happen here.
2. $acq^T_{\Gamma(x)} \preceq acq^{T'}_{\Gamma(x)} \preceq e^T_s \preceq e^{T'}_t$. This is a contradiction for the same reason: we know that the lock $\Gamma(x)$ is continually held from $acq^T_{\Gamma(x)}$ until e^T_t, so therefore $acq^{T'}_{\Gamma(x)}$ cannot happen here.
3. $acq^T_{\Gamma(x)} \preceq e^T_s \preceq acq^{T'}_{\Gamma(x)} \preceq e^{T'}_t$. This is the interesting case. It obeys all the constraints we know so far, however, the lock $\Gamma(x)$ is still held by thread T until released. The only way for these four events to occur in this order is for there to be a *fifth* event in thread T that releases the lock:

$$acq^T_{\Gamma(x)} \preceq e^T_s \preceq rel^T_{\Gamma(x)} \preceq acq^{T'}_{\Gamma(x)} \preceq e^{T'}_t$$

   where $\lambda(rel^T_{\Gamma(x)}) = t^T_{rel\ \Gamma(x)}$. Now we have precisely the events required by our definition of sufficient race conditions: the last four events above.

4. $acq^{T}_{\Gamma(x)} \preceq e^{T}_{s} \preceq e^{T'}_{t} \preceq acq^{T'}_{\Gamma(x)}$. Again we have a contradiction: we know $acq^{T'}_{\Gamma(x)} \preceq e^{T'}_{t}$.

Therefore, whenever our race detector concludes that two accesses are race-free, we are guaranteed two events, one in each thread, that happen between the two variable accesses, as required by our definition of sufficient race detectors. □

5.5 Pure-causal atomicity of sequenced statements

Lemma 16. *Assume a program $P \in$ Pure-Sml such that $\vdash P : ok$. Consider two statements (or expressions) s_1 and s_2 found in P, both in the same thread T, and let $N_1 = \text{TRANS}(s_1)$ and $N_2 = \text{TRANS}(s_2)$ be subnets of $\text{TRANS}(P)$ that result from the translation of the two statements. Assume that:*

1. *$\Gamma \vdash s_1 : a_1 \uparrow b_1$ and $\Gamma \vdash s_2 : a_2 \uparrow b_2$, and $(a_1; a_2) \sqsubseteq A$, $(a_1; b_2) \sqsubseteq A$, and $b_1 \sqsubseteq A$.*
2. *N_1 and N_2 are pure-causally atomic within $\text{TRANS}(P)$.*
3. *$\forall t_1 \in \text{END}(s_1).(t_1 \neq t_{break} \implies \exists t_2 \in \text{START}(s_2).t_1^{\bullet} \cap {}^{\bullet}t_2 \neq \emptyset)$ in the full Petri net, that is, that N_2 is "immediately after" N_1 in $\text{TRANS}(P)$ when s_1 terminates normally.*

Then $N_1 \cup N_2$ is pure-causally atomic.

Proof. We want to show that for every possible non-pure trace,

$$\nexists e^{T}_{2} \in \text{TRANS}(s_1) \cup \text{TRANS}(s_2), h^{T'''} \in \mathcal{E}. \quad (e_{start} \preceq h \preceq e_2) \wedge \nexists e^{T}_{end} \in_{\lambda} \text{END}(s_1; s_2).(e_{start} \preceq e_{end} \preceq e_2)$$

where as above, $e^{T}_{start} \in_{\lambda} \text{START}(s_1; s_2)$ and $e^{T}_{end} \in_{\lambda} \text{END}(s_1; s_2)$, and $h^{T'''}$ is an event in some thread $T''' \neq T$.

This is where our definition of causal purity is crucial: we are guaranteed that every `pure` block is causally pure (by our hypothesis that $\vdash P : ok$ and Lemma 10), which means that any possible event which can interfere with pure-causal atomicity (any impure action, and possibly any pure action on an abruptly-terminating trace) will exist in a non-pure trace, since it must be on a path that continues to a `break`.

Our condition is in nearly the exact same form as the definition of *causal atomicity*, except that here we are only considering a subset of all traces to find violations of pure-causal atomicity, and the ending set of events $\text{END}(s_1; s_2)$ is slightly different. It should seem intuitive, then, that pure-casual atomicity is more permissive than causal atomicity, since there are fewer opportunities to violate it. We can now essentially repeat the proof of Lemma 7, changing only the hypothesis to use the appropriate judgments:

By inspecting the operator (;) and by assumption 1, we see that $a_1 \sqsubseteq R$ or $a_2, b_2 \sqsubseteq L$ (or both). We have that s_1 and s_2 are pure-causally atomic by

assumption 2. From the definition of pure-causal atomicity, we therefore know that for any possible trace,

$$\nexists u_2^T \in \text{TRANS}(s_1), f^{T'} \in \mathcal{E}.(u_{start}^T \preceq f^{T'} \preceq u_2^T) \land \neg(u_{start}^T \preceq u_{end}^T \preceq u_2^T)$$
$$\nexists v_2^T \in \text{TRANS}(s_2), g^{T''} \in \mathcal{E}.(v_{start}^T \preceq g^{T''} \preceq v_2^T) \land \neg(v_{start}^T \preceq v_{end}^T \preceq v_2^T)$$

where $u_{start}^T \in START(s_1)$ and $u_{end}^T \in END(s_1)$ (resp. v_{start}^T and v_{end}^T) correspond to transitions in the START and END sets in the translation of s_1 (resp. s_2) and event $f^{T'}$ (resp. $g^{T''}$) is not in the same thread T.

Suppose that s_1 terminates abruptly. Then there are no events $e_2 \in_\lambda \text{TRANS}(s_2)$ at all, and so any trace that violates the pure-causal atomicity of $s_1; s_2$ must violate the pure-causal atomicity of s_1 alone, which is a contradiction. Therefore, s_1 must terminate normally, and s_2 must execute. This implies $\text{END}(s_1; s_2) = \text{END}(s_2)$. We do not care about the termination behavior of s_2, as we are guaranteed (since it is a non-pure trace) that any pure-causal atomicity violations we may observe are valid.

We therefore want to show that for every possible trace,

$$\nexists e_2^T \in \text{TRANS}(s_1 \cup \text{TRANS}(s_2)), h^{T'''} \in \mathcal{E}.$$
$$(e_{start} \preceq h \preceq e_2) \land \nexists e_{end}^T \in \text{END}(s_2).(e_{start} \preceq e_{end} \preceq e_2)$$

where as above, we define the events $e_{start}^T \in \text{START}(s_1)$ and $e_{end}^T \in \text{END}(s_2)$, and $h^{T'''}$ is an event in some thread $T''' \neq T$.

We proceed by contradiction. Suppose there existed some trace in which such an event $h^{T'''}$ did exist (and an event e_2^T that follows it). Such an event would satisfy

$$(e_{start} \preceq h \preceq e_2) \land \neg(e_{start} \preceq e_{end} \preceq e_2)$$

We know that $\text{LAST}(s_1, h)$ is not empty, because by assumption, $e_{start} \preceq h$, so it must be that either $e_{start} \in LAST(s_1, h)$ or else some other event e_L^T exists such that $e_{start} \preceq e_L \preceq h$, and $e_L \in \text{LAST}(s_1, h)$.

We know that $\text{FIRST}(s_1, h)$ is empty, because otherwise we could violate the atomicity of $\text{TRANS}(s_1)$:

$$\exists e_F^T \in \text{FIRST}(s_1, h).e_{start} \preceq h \preceq e_F$$

which is a contradiction. So h interacts with something in $\text{LAST}(s_1, h)$ and nothing after it in $\text{TRANS}(s_1)$.

We know that e_2 must be in $\text{TRANS}(s_2)$, since we know the set $\text{FIRST}(s_1, h)$ is empty, and no other events exist besides those in $\text{TRANS}(s_1)$ and $\text{TRANS}(s_2)$. Therefore, either $e_2 \in \text{FIRST}(s_2, h)$ or else some other event e_F^T exists such that $h \preceq e_F \preceq e_2$, and hence either way, $\text{FIRST}(s_2, h)$ is not empty.

We know that $\text{LAST}(s_2, h)$ is empty, because otherwise we could violate the atomicity of $\text{TRANS}(s_2)$:

$$\exists e_L^T \in \text{LAST}(s_2, h).e_L \preceq h \preceq e_2$$

which is a contradiction. So h interacts with something in $\text{LAST}(s_2, h)$ and nothing before it in $\text{TRANS}(s_2)$.

Therefore it must be that

$$\exists u^T \in \text{LAST}(s_1, h), v^T \in \text{FIRST}(s_2, h).u \preceq h \preceq v$$

In other words, if an event h exists at all, it must happen between the last interfering event in $\text{TRANS}(s_1)$ and the first interfering event in $\text{TRANS}(s_2)$ (based on the two sets above that are non-empty). Additionally

$$\nexists w \in \text{TRANS}(s_1 \cup \text{TRANS}(s_2)). \\ \forall u^T \in \text{LAST}(s_1, h), v^T \in \text{FIRST}(s_2, h).u \preceq w \preceq v$$

In other words, there are no other interfering events that matter in the translations of the two statements between those identified before (based on the two sets above that are empty).

It is possible that h is not unique, that is, there might be multiple events causally between u and v (for any pair of events u and v as above). Therefore, define events $h_1^{T'''}$ and $h_2^{T'''}$, not necessarily distinct from $h^{T'''}$, such that

$$u^T \prec h_1^{T'''} \preceq h^{T'''} \preceq h_2^{T'''} \prec v^T$$

Now we have our contradiction. Suppose s_1 is a right-mover. Then the event u must either be a lock acquire or a race-free variable access (by Lemma 11 above), and so must event h_1 (since $u \prec h_1$, which can only happen if both events access the same resource).

If both events are lock acquires of some lock l, then we have an immediate contradiction, since it is not possible for two threads to acquire the same lock simultaneously. In our translation into Petri nets, a lock-acquire operation moves a mark from l_{open} to l_T. Any other lock-acquire operations on the same lock l are not enabled until the lock is released and the mark is restored to l_{open}. However, we have that $u \prec h_1$, so no lock-release event occurs before h_1 supposedly happens. (A similar contradiction exists if h_1 is a lock release; thread T''' can't release a lock held by thread T.)

If both events access a variable (and clearly one must be a variable write, or else there is no conflict) then we appeal to the race-freedom of the access. We assumed the race detector was sound, so therefore both events must be race free. We also assumed that it was sufficient, so we know some lock l must be held for all accesses to this variable, or else there may be a race condition. So there must be some event in thread T which acquires lock l, and similarly there must be some event in thread T''' that acquires the same lock. Since we assumed that $u \prec h_1$, and u doesn't release the lock (it's a variable access, and does nothing to locks), h_1 must happen after the lock is released. We assumed that s_1 is a right mover, which means it cannot release the lock. So any further events that interact with lock l must occur in $\text{TRANS}(s_2)$ (by assumption 3, since $\text{TRANS}(s_2)$ is "immediately after" $\text{TRANS}(s_1)$ in the overall net—no other events in thread

T can intervene). Clearly, the first such event must be to release the lock (since it is currently held), and this could happen before or after event v. But the lock must be held for event v, since it is a race-free access, so if the lock *were* released, it must be reacquired before event v. But this is not atomic, which contradicts our assumption that s_2 effect-checked as atomic. Therefore, the lock l must be continuously held between u and v. We must have therefore that $v \preceq h_1$, but this contradicts our assumption that $h \preceq v$. We can conclude that if s_1 is a right-mover, then no event h can occur during s_1, between s_1 and s_2, or during s_2, and therefore that the union $N_1 \cup N_2$ is causally atomic.

A symmetric argument holds when s_2 is a left-mover. □

5.6 Pure-causal atomicity encompasses effect-based abstract atomicity

We can now state the main theorem of this section, which shows that pure-causal atomicity is at least as strong as effect-based pure-atomicity.

Theorem 2. *For every program $P \in$ Pure-SML where $\vdash P : ok$, then all* `atomic` *blocks in P are pure-causally atomic when translated into Petri nets.*

Proof. Consider an arbitrary expression $e \in P$ within an arbitrary atomic block. Expressions never terminate abruptly; for all intents and purposes they implicitly have an abrupt atomicity of $\bot$. Further, either e is entirely contained within a `pure` block or it is not: trivially, expressions cannot themselves contain `pure` blocks (which are statements), so all events in TRANS(e) are either causally within a `pure` block or not. We consider the two cases:

- If it is not contained within a `pure` block, then we know that $\Gamma, \Sigma \vdash e : a$ and $a \sqsubseteq A$, by the rules for atomic blocks. By Theorem 1 and Lemma 8, we are done.
- If it is in a `pure` block, then we know that $\Gamma, \Sigma \vdash e : a$ and $a \sqsubseteq A$ by the rules for pure blocks. By the same reasoning as above, we conclude that e is causally atomic, and therefore pure-causally atomic.

Consider an arbitrary statement $s \in P$ within an arbitrary atomic block. We assume that all of P effect-checks, including all `atomic` and `pure` blocks. We wish to show that if $\Gamma \vdash s : a \uparrow b$ and $\Gamma, X, p, \Sigma \vdash s : p', \Sigma'$, and $a, b \sqsubseteq A$, then TRANS(s) is pure-causally atomic in the net resulting from the translation of P.

- Case STMT-SKIP: $s =$ `skip`. Trivially, the translation of this statement is pure-causally atomic: there is only one transition in its translation.
- Case STMT-BREAK: $s =$ `break`. Trivially, the translation of this statement is pure-causally atomic: there is only one transition in its translation.
- Case STMT-SEQ: $s = s_1; s_2$. We know that $\Gamma \vdash s : a \uparrow b$ and $a, b \sqsubseteq A$. We therefore know that $\Gamma \vdash s_1 : a_1 \uparrow b_1$, $\Gamma \vdash s_2 : a_2 \uparrow b_2$, and $(a_1; a_2) \sqsubseteq A$ and $(b_1 \sqcup (a_1; b_2)) \sqsubseteq A$ by the typing rules. This implies

that $a_1, a_2, b_1, b_2, (a_1; b_2) \sqsubseteq A$. We have from induction that TRANS(s_1) and TRANS(s_2) are pure-causally atomic. This meets all the criteria for Lemma 16, and so we are done.

– Case STMT-LOOP: $s = \texttt{loop}\ s'$. We know that s' is pure-causally atomic by induction; it remains to show that the loop is pure-causally atomic. However, the translation of `loop`s merely sequences the loop body with itself, hence using the preceding lemma, we are done.
– Case STMT-PURE: $s = \texttt{pure}\ s'$. We know from Lemma 10 that s' is causally pure. From the typing rule, we know $\Gamma \vdash \texttt{pure}\ s' : B \uparrow b$, $\Gamma \vdash s' : a' \uparrow b$, and $a' \sqsubseteq A$. But by our hypothesis that $\vdash P : ok$, we also know $b \sqsubseteq A$. Therefore we can conclude that s' is pure-causally atomic. Looking at TRANS(s) $\setminus$ TRANS(s'), we see that we construct many transitions (the snapshots and the checkpoints, and administrative transitions), all of which are connected only to places in the current thread, to the locks, or to the place LockSnap; in other words, they induce no causal dependencies with any other threads. Therefore, if there existed a trace that violated the pure-causal atomicity of s, it must violate the pure-causal atomicity of s', which is impossible.
– Case STMT-ASSIGN and STMT-ASSIGN-RACE: $s = x := exp$. The translation of this expression decomposes into two subnets: the TRANS(exp), and a transition t_x^T that accesses the variable x. We know that TRANS(e) is pure-causally atomic. Therefore, if there exists an event $f^{T'}$ that witnesses the non-atomicity of TRANS(s), it must satisfy

$$\exists e_1^T \in_\lambda \text{START}(s), e_2^T \in_\lambda \mathcal{E}.\quad (e_1 \preceq f \preceq e_2) \wedge (\nexists e_{end}^T \in_\lambda \text{END}(s).e_1 \preceq e_{end} \preceq e_2)$$

By examining TRANS(s), we see that END(s) $= \{t_x^T\}$ and START(s) $=$ START(exp). Moreover, we know that $e_2 \notin$ CURRENT(exp, e_1), since if it were, then $\nexists e_{end} \in$ END(exp).$e_{end} \preceq e_2$, and then f would witness the non-atomicity of TRANS(exp), which is impossible by assumption. Therefore we must have CURRENT(s, e_1) $\setminus$ CURRENT(exp, e_1) $= \{e_2\}$. But then $\exists e_{end} \in$ END(s).$e_{end} \preceq e_2$, which contradicts our assumption about f.
– Case STMT-IF: $s = \texttt{if}\ exp\ s_1\ s_2$. The translation of s decomposes into three subnets, one per subexpression, as well as two extra transitions t_t and t_f indicating which branch of the `if` to take. By induction, we assume that each of these subnets is pure-causally atomic. It is trivial to see that "grouping" t_t with TRANS(s_1) and t_f with TRANS(s_2) does not affect their pure-causal atomicity: since these transitions are immediately before only the entry points of TRANS(s_1) and TRANS(s_2), any event f that could witness the non-atomicity of the grouping would witness the non-atomicity of the subexpressions themselves. By assumption, $\Gamma \vdash s : a \uparrow b$ and $a, b \sqsubseteq A$. Following the inference rule, we have $\Gamma \vdash e : a_e$ and $\Gamma \vdash s_i : a_i \uparrow b_i$. We therefore know that $(a_e; (a_1 \sqcup a_2)) \sqsubseteq a$ and $(a_e; (b_1 \sqcup b_2)) \sqsubseteq a$. This meets all the requirements for Lemma 16, so we apply it twice, with e and s_1 and then with e and s_2, and so we are done.

– Case Stmt-Atomic: $s = \texttt{atomic}\ s'$. We know TRANS(s) = TRANS(s'), so by induction we are done.
– Case Stmt-Acquire and Stmt-Release: $s = \texttt{acquire}\ l$ or $s = \texttt{release}\ l$. Trivially, the translation of this statement is pure-causally atomic: there is only one transition in its translation.
– Case Stmt-Block: $s = \texttt{block}\ s'$. We know that s' is pure-causally atomic by induction. Examining TRANS(s) \ TRANS(s'), we see that we construct one new transition t_{out} to handle any `break` statements within s', and connect each of them to this t_{out}. We therefore know that if t_{out} fires, that $e_{break} \prec e_{out} \prec e_{in}$, where $\lambda(e_{break}) = t_{break}$ is the event when the `break` statement executes; $\lambda(e_{out}) = t_{out}$ is the event of t_{out} firing, and e_{in} is the next event in this thread (the start of the statement following s). So if there existed an event f that violated the atomicity of s, such that $e_{begin} \preceq f \preceq e_{out}$, it must be that $e_{begin} \preceq f \preceq e_{break}$, which violates the atomicity of s', leading to a contradiction. Hence s is pure-causally atomic as well.

□

6 Implementing the definition of pure-causal atomicity

As in [1], we define four primitive colors describing the atomicity state:

– Achromatic (A) — default "non"-color
– Blue (B) — begin atomic block
– Yellow (Y) — another thread saw a blue mark
– Red (R) — atomicity or purity was violated

We also define four primitive colors describing the purity state:

– Not pure (n) — default "non"-color
– Pure ok (o) — while checking purity, beginning a pure block, without yet performing a mutation
– Pure bad (b) — while checking purity, within a pure block, but have performed a mutation
– Must break (m) — while checking atomicity, within a pure block, and must therefore break

Our colors will be the set of pairs $\{A, B, Y, R\} \times \{n, o, b, m\}$. We define a lattice structure over these component-wise, with $A \sqsubset B \sqsubset Y \sqsubset R$ and $n \sqsubset o \sqsubset b \sqsubset m$. Define the projection functions atom(x) and pure(x) to extract the first and second components of a color, respectively. To define the color transitions, we need rules that are backwards-compatible with the existing approach: if a program doesn't use any pure blocks, the analysis ought to be identical to that in [1]. We note that each widget defined in our translation uses a "program counter," a token that passes from place to place and uniquely identifies the current execution point of the current thread. (This place was named p_{in} in figure 4 of [1].) We distinguish this input as special, in the translation rules that

follow, and denote by pc_t the current program counter of thread t; there are T threads in total.

Accordingly, we define the following "boring" rules:

- In the OR widget, each transition shifts pc_t from input to output unchanged, and shifts the token from the active input place to the output place unchanged. These two tokens could be of different colors.
- On lock acquire and release operations, the token is moved from L_i-*other* to L_i-*held* (or vice versa) unchanged.
- The SNAPSHOT widget leaves its input token colors unchanged.
- The CHECK widget leaves the pc_t color unchanged, and may place (R, n) on ERR. The anonymous transitions leave their inputs unchanged; the SNAPSHOT widget leaves its inputs untouched. The two successful AND transitions produce the result of the pc_t input; the two error AND transitions produce (R, n).
- All bookkeeping transitions leave their input colors alone.

All other transitions are interesting, in that they involve interactions with the variable or lock places. For these we define the following rules:

- When entering a `pure` block, we have three mutually-exclusive choices of paths. We can choose to skip over the entire `pure` block, jumping directly to the end of the block and continuing execution. This models executions where the pure block terminates normally or never terminates, and produces no side effects; we therefore leave the color at n and continue at the end of the block. We can choose to check purity, tracing through the block and ensuring that all writes are post-dominated by a `break` and all lock manipulations are undone on normal exit. To do this we set the color to o, and continue through the block. Finally, we can choose to check atomicity, tracing through the block and checking for interactions with other threads. To do this we set the color to m. If the `pure` block never breaks on a particular trace, then we simply "get stuck"; instead we should have chosen to model the behavior of the block as a skip.
- On variable reads in thread t, the purity value for v_t and pc_t is unchanged.
- On variable writes in thread t, we update the purity value for v_t and for pc_t:

$$\text{pure}(pc_t) := \text{pure}(v_t) := \begin{cases} b & \text{if } \text{pure}(v_t) = o \vee \text{pure}(pc_t) = o \\ pc_t & \text{otherwise} \end{cases}$$

  The purity colors of other variables (v_k where $k \neq t$) are unchanged.
- On lock manipulations, writes and reads, we update the colors of all inputs: If $\text{pure}(pc_t) = b$ or $\text{pure}(pc_t) = o$, then we leave the colors alone, as we're not checking for atomicity. Otherwise, we use the same coloring rules as for the causal atomicity case.
- The last remaining special case is the transitions exiting a `pure` block. If we are in purity-checking mode, then we require that the color be o on normal exit; on both normal and abrupt exit we clear the mark on ERR. This

ensures that no further atomicity checking is attempted on this trace, since we may have missed an atomicity violation while examining this block solely for purity. If we are in atomicity-checking mode, however, then we get stuck on normal termination, and only abrupt termination can continue execution. (The transition that skips the entire block bypasses this restriction, so we can model both behaviors of the `pure` block correctly.)
– We add a transition $v_k@(R, n) \rightarrow ERR$, for every variable place v_k (i.e. every variable v and every thread k). (More precisely, we add transitions with this effect, but that are 1-safe.)

Checking for atomicity and purity simultaneously simply asks whether the ERROR transition is reachable with a red mark.

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. IEEE Transactions on Software Engineering **31**(4) (2005) 275–291
3. 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