Thread-Local Semantics and its Efficient Sequential Abstractions for Race-Free Programs

Suvam Mukherjee[1], Oded Padon[2], Sharon Shoham[2], Deepak D'Souza[1], and Noam Rinetzky[2]

[1] Indian Institute of Science, India
[2] Tel Aviv University, Israel

Abstract. Data race free (DRF) programs constitute an important class of concurrent programs. In this paper we provide a framework for designing and proving the correctness of data flow analyses that target this class of programs, and which are in the same spirit as the "sync-CFG" analysis originally proposed in [9]. To achieve this, we first propose a novel concrete semantics for DRF programs called *L-DRF* that is *thread-local* in nature with each thread operating on its own copy of the data state. We show that abstractions of our semantics allow us to reduce the analysis of DRF programs to a *sequential* analysis. This aids in rapidly porting existing sequential analyses to scalable analyses for DRF programs. Next, we parameterize the semantics with a partitioning of the program variables into "regions" which are accessed atomically. Abstractions of the region-parameterized semantics yield more precise analyses for region-race free concurrent programs. We instantiate these abstractions to devise efficient relational analyses for race free programs, which we have implemented in a prototype tool called RATCOP. On the benchmarks, RATCOP was able to prove upto 65% of the assertions, in comparison to 25% proved by a version of the analysis from [9].

1 Introduction

Our aim in this paper is to provide a framework for developing data-flow analyses which specifically target the class of data race free (DRF) concurrent programs. The starting point of this work is the so-called "sync-CFG" style of analysis proposed in [9] for race-free programs. The analysis here essentially runs a sequential analysis on each thread, communicating data-flow facts between threads only via "synchronization edges" that go from a release statement in one thread to a corresponding acquire statement in another thread. The analysis thus runs on the control-flow graphs (CFGs) of the threads augmented with synchronization edges, as shown in the center of Fig. 1, which explains the name for this style of analysis. The analysis computes data flow facts about the value of a variable that are sound *only* at points where that variable is *relevant*, in that it is read or written to at that point. The analysis thus trades unsoundness of facts at irrelevant points for the efficiency gained by restricting interference between threads to points of synchronization alone.

However, the analysis proposed in [9] suffers from some drawbacks. Firstly, the analysis is intrinsically a "value-set" analysis, which can only keep track of the set of values each variable can assume, and not the relationships *between* variables. Any naive attempt to extend the analysis to a more precise relational one quickly leads to unsoundness. The second issue is to do with the technique for establishing soundness. A convenient way to prove soundness of an analysis is to show that it is a consistent abstraction [7] of a canonical analysis, like the collecting semantics for sequential programs or the interleaving semantics for concurrent programs. However, a sync-CFG style analysis *cannot* be shown to be a consistent abstraction of the standard interleaving semantics, due largely to the unsoundness at irrelevant points. Instead, one needs to use an intricate argument, as done in [9], which essentially shows that in the least fixed point of the analysis, every write to a variable will flow to a read of that variable via a happens-before path (that is guaranteed to exist by the property of race-freedom). Thus, while one can argue soundness of an analysis that abstracts the value-set analysis by showing it to be a consistent abstraction of the value set analysis, to argue soundness of any other proposed sync-CFG style analysis (in particular one that is more precise than the value-set analysis), one would have to resort to a similar involved proof as in [9].

Towards addressing these issues, we propose a framework that facilitates the design of different sync-CFG analyses with varying degrees of precision and efficiency. The foundation of this framework is a thread-local semantics for DRF programs, which can play the role of a “most precise” analysis which other sync-CFG analyses can be shown to be consistent abstractions of. This semantics, which we call *L-DRF*, is similar to the interleaving semantics of concurrent programs [20], but keeps thread-local (or per-thread) copies of the shared state. Intuitively, our semantics works as follows. Apart from its local copy of the shared data state, each thread t also maintains a per-variable version count, which is incremented whenever t updates the variable. The exchange of information between threads is via buffers, associated with release points in the program. When a thread releases a lock, it stores its data state to the corresponding buffer, along with the version counts of the variables. As a result, the buffer of a release point records both the local data state and the variable versions as they were when the release was last executed. When some thread t acquires a lock m, it compares its per-variable version count with those in the buffers pertaining to release points associated with m, and copies over the valuation of a variable to its local state, if it is newer in some buffer (as indicated by a higher version count). Similar to a sync-CFG analysis, the value of a shared variable in the local state of a thread may be *stale*. *L-DRF* leverages the race freedom property to ensure that the value of a variable is correct in a local state at program points where it is *read*. It thus captures the essence of a sync-CFG analysis. The *L-DRF* semantics is also of independent interest, since it can be viewed as an alternative characterization of the behavior of data race free programs.

The analysis induced by the *L-DRF* semantics is shown to be sound for DRF programs. In addition, the analysis is in a sense the most precise sync-CFG analysis one can hope for, since at every point in a thread, the relevant part of the thread-local copy of the shared state is *guaranteed* to arise in some execution of the program.

Using the *L-DRF* semantics as a basis, we now propose several precise and efficient *relational* sync-CFG analyses. The soundness of these analyses all follow immediately, since they can easily be shown to be consistent abstractions of the *L-DRF* analysis. The key idea behind obtaining a sound relational analysis is suggested by the *L-DRF* analysis: at each `acquire` point we apply a *mix* operator on the abstract values, which essentially amounts to forgetting all correlations between the variables.

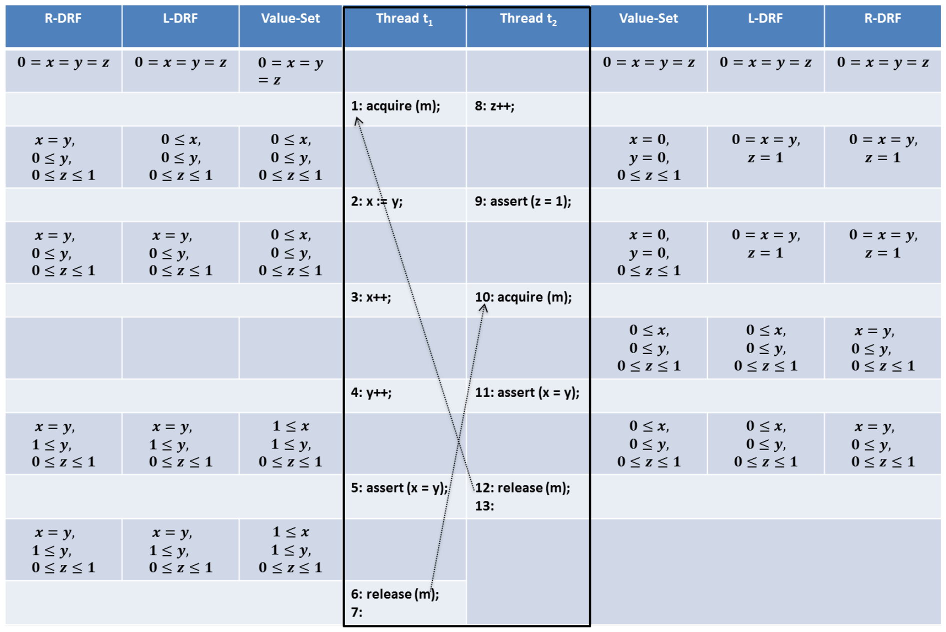
While these analyses allow maintaining fully-relational properties within thread-local states, communicating information over cross-thread edges loses all correlations due to the *mix* operation. To improve precision further, we refine the *L-DRF* semantics to take into account *data regions*. Technically, we introduce the notion of *region race freedom* and develop the *R-DRF* semantics: the programmer can partition the program variables into "regions" that should be accessed *atomically*. A program is *region race free* if it does not contain conflicting accesses to variables in the same region, that are unordered by the happens-before relation. The classical notion of data race freedom is a special case of region race freedom where each region consists of a single variable, and techniques to determine that a program is race free can be naturally extended to determine region race freedom (see Section 6). For region race free programs, *R-DRF*, which refines *L-DRF* by taking into account the atomic nature of accesses that the program makes to variables in the same region, produces executions which are indistinguishable, with respect to reads of the regions, from the ones produced by *L-DRF*. By leveraging the *R-DRF* semantics as a starting point, we obtain more precise sequential analyses that track relational properties *within regions* across threads. This is obtained by refining the granularity of the *mix* operator from single variables to regions.

We have implemented our analyses in a prototype analyzer called RATCOP, and provide a thorough empirical evaluation in Sec. 7. We show that RATCOP attains a precision of up to 65% on a subset of race-free programs from the SV-COMP15 suite. In contrast, an interval based value-set analysis derived from [9] was able to prove only 25% of the assertions. On a separate set of experiments, RATCOP turns out to be nearly 5 orders of magnitude faster than an existing state-of-the-art abstract interpretation based tool [25].

2 Overview

We illustrate the *L-DRF* semantics, and its sequential abstractions, on the simple program in Fig. 1. We assume that all variables are shared and are initialized to 0. The threads access x and y only after acquiring lock m. The program is free from data races.

A state in the *L-DRF* semantics keeps track of the following components: a location map pc mapping each thread to the location of the next command to be executed, a lock map μ which maps each lock to the thread holding it, a local environment (variable to value map) Θ for each thread, and a function Λ which maps each buffer (associated with each location following a release command) to an environment. Every release point of each lock m has an associated buffer, where a thread stores a copy of its local environment when it executes the corresponding release instruction. In the environments, each variable x has a version count associated with it which, along any execution π, essentially associates this valuation of x with a unique prior write to it in π. As an example, the "versioned" environment $\langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 0^0 \rangle$ says that x and y have the value 1 by the 1^{st} writes to x and y, and z has not been written to. An execution is an interleaving of commands from the different threads. Consider an execution where, after a certain number of steps, we have the state $pc(t_1 \mapsto 6, t_2 \mapsto 10), \Theta(t1) = \langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 0^0 \rangle, \Theta(t2) = \langle x \mapsto 0^0, y \mapsto 0^0, z \mapsto 1^1 \rangle, \mu(m) = t_1, \Lambda = \bot$. The buffers are all empty as no thread has executed a `release` yet. Note that the values (and ver-



R-DRF	L-DRF	Value-Set	Thread t_1	Thread t_2	Value-Set	L-DRF	R-DRF
$0 = x = y = z$	$0 = x = y = z$	$0 = x = y = z$			$0 = x = y = z$	$0 = x = y = z$	$0 = x = y = z$
			1: acquire (m);	8: z++;			
$x = y$, $0 \leq y$, $0 \leq z \leq 1$	$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$	$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$			$x = 0$, $y = 0$, $0 \leq z \leq 1$	$0 = x = y$, $z = 1$	$0 = x = y$, $z = 1$
			2: x := y;	9: assert (z = 1);			
$x = y$, $0 \leq y$, $0 \leq z \leq 1$	$x = y$, $0 \leq y$, $0 \leq z \leq 1$	$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$			$x = 0$, $y = 0$, $0 \leq z \leq 1$	$0 = x = y$, $z = 1$	$0 = x = y$, $z = 1$
			3: x++;	10: acquire (m);			
					$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$	$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$	$x = y$, $0 \leq y$, $0 \leq z \leq 1$
			4: y++;	11: assert (x = y);			
$x = y$, $1 \leq y$, $0 \leq z \leq 1$	$x = y$, $1 \leq y$, $0 \leq z \leq 1$	$1 \leq x$ $1 \leq y$, $0 \leq z \leq 1$			$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$	$0 \leq x$, $0 \leq y$, $0 \leq z \leq 1$	$x = y$, $0 \leq y$, $0 \leq z \leq 1$
			5: assert (x = y);	12: release (m); 13:			
$x = y$, $1 \leq y$, $0 \leq z \leq 1$	$x = y$, $1 \leq y$, $0 \leq z \leq 1$	$1 \leq x$ $1 \leq y$, $0 \leq z \leq 1$					
			6: release (m); 7:				

Fig. 1: A simple race free program with two threads t_1 and t_2, with all variables being shared and initialized to 0. The columns *L-DRF* and *R-DRF* show the facts computed by polyhedral abstractions of our thread-local semantics and its region-parameterized version, respectively. The Value-Set column shows the facts computed by interval abstractions of the Value-Set analysis of [9]. *R-DRF* is able to prove all 3 assertions, while *L-DRF* fails to prove the assertion at line 11. Value-Set only manages to prove the simple assertion at line 9.

sions) of x and y in $\Theta(t_2)$ are *stale*, since it was t_1 which last modified them (similarly for z in $\Theta(t_1)$). Next, t_1 can execute the `release` at line 6, thereby setting $\mu(m) = \_$ and storing its current local state to $\Lambda(7)$. Now t_2 can execute the `acquire` at line 10. The state now becomes $pc(t_1 \mapsto 7, t_2 \mapsto 11)$, $\mu(m) = t_2$, and t_2 now "imports" the most up-to-date values (and versions) of the x and y from $\Lambda(7)$. This results in its local state becoming $\langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 1^1 \rangle$ (the valuations of x and y are pulled in from the buffer, while the valuation of z in t_2's local state persists). The value of x and y in $\Theta(t_2)$ is no longer stale: *L-DRF* leveraged the race freedom to ensure that the values of x and y are correct when they are read at line 11.

Roughly, we obtain *sequential* abstractions of *L-DRF* via the following steps: (i.) Provide a data abstraction of sets of environments (ii.) Define the state to be a map from locations to these abstract data values (iii.) Draw inter-thread edges by connecting releases and acquires of the same lock (as shown in Fig. 1) (iv.) Define an abstract *mix* operation which soundly approximates the "import" step outlined earlier (v.) Analyze the program as if it was a sequential program, with *inter*-thread join points (the `acquire`'s) using the *mix* operator.

The analysis in [9] is precisely such a sequential abstraction, where the abstract data values are abstractions of *value-sets* (variables mapped to sets of values). Value sets do not track correlations between variables, and only allow coarse abstractions like

Intervals [6]. The *mix* operator, in this case, turns out to be the standard join. For Fig. 1, the interval analysis only manages to prove the assertion at line 9.

A more precise relational abstraction of *L-DRF* can be obtained by abstracting the environments as, say, convex polyhedra [8]. As shown in Fig. 1, the resulting analysis is more precise than the interval analysis, being able to prove the assertions at lines 5 and 9. However, in this case, the *mix* must forget the correlations among variables in the incoming states: it essentially treats them as value sets. This is essential for soundness. Thus, even though the `acquire` at line 10 obtains the fact that $x = y$ from the buffer at 7, and the incoming fact from 9 also has $x = y$, it fails to maintain this correlation after the *mix*. Consequently, it fails to prove the assertion at line 11.

Finally, one can exploit the fact that x and y form a data region, that is always accessed atomically by the two threads. The program is thus *region race free*, for this particular region definition. One can parameterize the *L-DRF* semantics with this region definition, to yield the *R-DRF* semantics. The resulting sequential abstraction maintains relational information as in polyhedra based analysis derived from *L-DRF*, but has a more precise *mix* operator which preserves relational facts which hold *within* a region. Since both the incoming facts at line 10 satisfy $x = y$, the *mix* preserves this fact, and the analysis is able to prove the assertion at 11.

Note that in all the three analyses, we are guaranteed to compute sound facts for variables *only* at points where they are accessed. For example, all three analyses claim that x and y are both 0 at line 9, which is clearly wrong. However, x and y are not accessed at this point. We make this trade-off for the soundness guarantee in order to achieve a more efficient analysis. Also note that in Figure 1, the inter-thread edges add a spurious loop in the program graph (and, therefore, in the analysis of the program), which prevents us from computing an upper bound for the values of x and y. We show in a later section how we can appropriately abstract the versions to avoid some of these spurious loops.

3 Preliminaries

Mathematical Notations. We use $\rightarrow$ and $\rightharpoonup$ to denote total and partial functions, respectively, and $\perp$ to denote a function which is not defined anywhere. We use $\_$ to denote an irrelevant value which is implicitly existentially quantified. We write $\bar{S}$ to denote a (possibly empty) finite sequence of elements coming from a set S. We denote the *length* of a sequence π by $|\pi|$, and the i-th element of π, for $0 \leq i < |\pi|$, by π_i. We denote the domain of a function ϕ by $\mathrm{dom}(\phi)$ and write $\phi[x \mapsto v]$ to denote the function $\lambda y. \mathit{if}\, y = x\, \mathit{then}\, v\, \mathit{else}\, \phi(y)$. Given a pair of function $\upsilon = \langle \phi, \nu \rangle$, we write $\upsilon\phi$ and $\upsilon\nu$ to denote ϕ and ν, respectively.

3.1 Programming Language and Programs

A multi-threaded program P consists of four finite sets: *threads* $\mathcal{T}$, *control locations* $\mathcal{L}$, program *variables* $\mathcal{V}$ and *locks* (*mutexes*) $\mathcal{M}$. We denote by $\mathbb{V}$ the set of values the program variables can assume. Without loss of generality, we assume in this work that $\mathbb{V}$ is simply the set of integers. Figure 2 lists the semantic domains we use in this paper and the metavariables ranging over them..

Type	**Syntax**	**Description**
Assignment	$x := e$	Assigns the value of expression e to variable $x \in \mathcal{V}$
Assume	$\texttt{assume}(b)$	Blocks the computation if boolean condition b does not hold
Acquire	$\texttt{acquire}(m)$	Acquires lock m, provided it is not *held* by any thread
Release	$\texttt{release}(m)$	Releases lock m, provided the executing thread holds it

Table 1: Program Commands

Every thread $t \in \mathcal{T}$ has an entry location ent_t and a set of instructions $inst_t \subseteq \mathcal{L} \times cmd \times \mathcal{L}$, which defines the *control flow graph* of t. An instruction $\langle n_s, c, n_t \rangle$ comprises a *source* location n_s, a *command* $c \in cmd$, and a *target* location n_t. The set of program commands, denoted by cmd, is defined in Table 1. (Commands like `fork` and `join` of a bounded number of threads can be simulated using locks.) For generality, we refrain from defining the syntax of the expressions e and boolean conditions b.

We denote the set of commands appearing in program P by $cmd(P)$. We refer to an assignment $x := e$ as a *write-access* to x, and as a *read-access* to every variable that appears in e. Without loss of generality, we assume variables appearing in conditions of $\texttt{assume}()$ commands in instructions of some thread t do not appear in any instruction of any other thread $t' \neq t$.

We denote by $\mathcal{L}_t$ the set of locations in instructions of thread t, and require that the sets be disjoint for different threads. For a location $n \in \mathcal{L}$ $(= \bigcup_{t \in \mathcal{T}} \mathcal{L}_t)$, we denote by $tid(n)$ the thread t which contains location n, i.e., $n \in \mathcal{L}_t$. We forbid different instructions from having the same source and target locations, and further expect instructions pertaining to assignments, $\texttt{acquire}()$ and $\texttt{release}()$ commands to have unique source and target locations. Let $\mathcal{L}_t^{rel}$ be the set of program locations in the body of thread t following a $\texttt{release}()$ command. We refer to $\mathcal{L}_t^{rel}$ as t's *post-release points* and denote the set of *release points* in a program by $\mathcal{L}^{rel} = \bigcup_{t \in \mathcal{T}} \mathcal{L}_t^{rel}$. Similarly, we define t's *pre-acquire points*, denoted by $\mathcal{L}_t^{acq}$, and denote a program's *acquire points* by $\mathcal{L}^{acq} = \bigcup_{t \in \mathcal{T}} \mathcal{L}_t^{acq}$. We denote the sets of post-release and pre-acquire points pertaining to operations on lock m by $\mathcal{L}_m^{rel}$ and $\mathcal{L}_m^{acq}$, respectively.

3.2 Standard Interleaving Semantics

Let us fix a program $P = (\mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{M})$ for the rest of this paper. We define the standard interleaving semantics of a program using a labeled transition system $\langle \mathcal{S}, s_{ent}, TR^{\mathrm{s}} \rangle$, where $\mathcal{S}$ is the set of *states*, $s_{ent} \in \mathcal{S}$ is the *initial state*, and $TR^{\mathrm{s}} \subseteq \mathcal{S} \times \mathcal{T} \times \mathcal{S}$ is a transition relation, as defined below.

$t \in \mathcal{T}$	Thread identifiers	$pc \in PC$	$\equiv \mathcal{T} \to \mathcal{L}$	Program counters
$n \in \mathcal{L}$	Program locations	$\mu \in LM$	$\equiv \mathcal{M} \rightharpoonup \mathcal{T}$	Lock map
$x, y \in \mathcal{V}$	Variable identifiers	$\phi \in Env$	$\equiv \mathcal{V} \to \mathbb{V}$	Environments
$l \in \mathcal{M}$	Lock identifiers	$\nu \in \mathcal{V}V$	$\equiv \mathcal{V} \to \mathbb{N}$	Variable versions
$r \in R$	Region identifiers	$\upsilon \in VE$	$\equiv Env \times \mathcal{V}V$	Versioned environments
$v \in \mathbb{V}$	Values			

$s = \langle pc, \mu, \phi \rangle \in \mathcal{S} \equiv PC \times LM \times Env$	Standard States
$\sigma = \langle pc, \mu, \Theta, \Lambda \rangle \in \Sigma \equiv PC \times LM \times (\mathcal{T} \to VE) \times (\mathcal{L} \to VE)$	Thread-Local States

Fig. 2: Semantic Domains.

States A *state* $s \in \mathcal{S}$ is a tuple $\langle pc, \mu, \phi \rangle$, where $pc \in PC \overset{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ records the *program counter* (or location) of every thread, $\mu \in LM \overset{\text{def}}{=} \mathcal{M} \rightharpoonup \mathcal{T}$ is a *lock map* which associates every lock to the thread that holds it (if such a thread exists), and $\phi \in Env \overset{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ is an *environment*, mapping variables to their values.

Initial State We refer to the state $s_{ent} = \langle \lambda t.\, ent_t, \bot, \lambda x.\, 0 \rangle$ where every thread is at its entry program location, no thread holds a lock, and all the variables are initialized to zero as the *initial state*.

Transition Relation The transition relation $TR_P^{\mathbf{s}} \subseteq \mathcal{S} \times \mathcal{T} \times \mathcal{S}$ captures the interleaving semantics of a program P. A transition $\tau = \langle s, t, s' \rangle$, also denoted by $\tau = s \to_t s'$, says that thread t can execute a command which transforms (the *source*) state s to (the *target*) state s'. As such, the transition relation is the set of all possible transitions generated by its commands, i.e. $TR_P^{\mathbf{s}} = \bigcup_{c \in cmd(P)} TR_c^{\mathbf{s}}$. In these transitions, one thread executes a command, and changes its program counter accordingly, while all other threads remain stationary. Due to space constraints, we omit the formal definitions of $TR_c^{\mathbf{s}}$, which is standard, and only provide a brief informal description. An assignment $x := e$ command updates the value of the variables according to the expression e. An $\texttt{assume}(b)$ command generates transitions only from states in which the boolean interpretation of the condition b is $True$. An $\texttt{acquire}(m)$ command executed by thread t sets $\mu(m) = t$, provided the lock m is not held by any other thread, A $\texttt{release}(m)$ command executed by thread t sets $\mu(m) = \_$, provided t holds m. A thread attempting to release a lock that it does not own gets stuck.[3]

Notations. For a transition $\tau = \langle pc, \mu, \phi \rangle \to_t \langle pc', \mu', \phi' \rangle \in TR_P^{\mathbf{s}}$, we denote by $t(\tau) = t$ the thread that *executes* the transition, and by $c(\tau)$ the (unique) command $c \in cmd(P)$, such that $\langle pc(t), c, pc'(t) \rangle \in inst_t$, which it executes. We denote by $n(\tau) = pc(t)$ and $n'(\tau) = pc'(t)$, the source and target locations of the executed instruction respectively.

Executions An execution π of a concurrent program P is a finite sequence of transitions coming from its transition relation, such that s_{ent} is the source of transition π_0 and the source state of every transition π_i, for $0 < i < |\pi|$, is the target state of transition π_{i-1}. By abuse of notation, we also write executions as sequences of states interleaved with thread identifiers: $\pi = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} s_n$.

Collecting semantics. The collecting semantics of a program P according to the standard semantics is the set of reachable states starting from the initial state s_{ent}:

$$[\![P]\!]^{\mathbf{s}} = LFP\ \lambda X.\, \{s_{ent}\} \cup \{s' \mid s \to_t s' \wedge s \in X \wedge t \in \mathcal{T}\}$$

3.3 Data Races and the Happens-Before Relation

We say that two commands *conflict* on a variable x, if they both access x, and at least one access is a write. A program contains a *data race* when two concurrent threads may execute conflicting commands, and the threads use no explicit mechanism to prevent their accesses from being simultaneous [29]. A program which has no data races is said to be *data race free*. A standard way to formalize the notion of *data race freedom*

[3] The decision to block a thread releasing a lock it does not own was made to simplify the semantics. Our results hold even if this action aborts the program.

(DRF), is to use the *happens before* [19] relation induced by executions. An execution is *racy* if it contains a pair of transitions executing conflicting commands which are not ordered according to the happens-before relation. A program which has no racy execution is said to be *data race free*.

For a given execution, the happens-before relation is defined as the reflexive and transitive closure of the *program-order* and *synchronizes-with* relations, formalized below.

Definition 1 (Program order). *Let π be an execution of P. Transition π_i is related to the transition π_j according to the* program-order *relation in π, denoted by $\pi_i \xrightarrow{po}_\pi \pi_j$, if $j = \min\{k \mid i < k < |\pi| \wedge t(\pi_k) = t(\pi_i)\}$, i.e., π_i and π_j are successive executions of commands by the same thread.*[4]

Definition 2 (Synchronize-with). *Let π be an execution of P. Transition π_i is related to the transition π_j according to* synchronizes-with *relation in π, denoted by $\pi_i \xrightarrow{sw}_\pi \pi_j$, if $c(\pi_i) = \texttt{release}(m)$ for some lock m, and $j = \min\{k \mid i < k < |\pi| \wedge c(\pi_k) = \texttt{acquire}(m)\ \}$, i.e., π_i and π_j are successive release and acquire commands of the same lock in the execution.*

Definition 3 (Happens before). *The* happens-before *relation pertaining to an execution π of P, denoted by $\cdot \xrightarrow{hb}_\pi \cdot$, is the reflexive and transitive closure of the union of the program-order and synchronizes-with relations induced by the execution π.*

Note that transitions executed by the same thread are always related according to the happens-before relation.

Definition 4 (Data Race). *Let π be an execution of P. Transitions π_i and π_j constitute a racing pair, or a data-race, if the following conditions are satisfied: (i) $c(\pi_i)$ and $c(\pi_j)$ both access the variable x, with at least one of the accesses being a write to x, and (ii) neither $\pi_i \xrightarrow{hb}_\pi \pi_j$ nor $\pi_j \xrightarrow{hb}_\pi \pi_i$ holds.*

4 Thread-local Semantics for Data-Race Free Programs (*L-DRF*)

In this section, we define a new thread-local semantics for datarace free concurrent programs, which we refer to as *L-DRF* semantics. The new semantics, like the standard one defined in Section 3, is based on the interleaving of transitions made by different threads, and the use of a lock map to coordinate the use of locks. However, unlike the standard semantics, where the threads share access to a single *global* environment, in the *L-DRF* semantics, every thread has its own *local* environment which it uses to evaluate conditions and perform assignments. Threads exchange information through *release buffers*: every post-release point $n \in \mathcal{L}_t^{rel}$ of a thread t is associated with a *buffer*, $\Lambda(n)$, which records a snapshot of t's local environment the last time t ended up at the program point n. Recall that this happens right after t executes the instruction

[4] Strictly speaking, the various relations we define are between indices $\{0, \ldots, |\pi| - 1\}$ of an execution, and not transitions, so we should have written, e.g., $i \xrightarrow{po}_\pi j$ instead of $\pi_i \xrightarrow{po}_\pi \pi_j$. We use the rather informal latter notation, for readability.

$\langle n_s, \texttt{release(m)}, n\rangle \in inst_t$. When a thread t acquires a lock m, it updates its local environment using the snapshots stored in the buffers pertaining to the release of m. To ensure that t updates its environment such that the value of every variable is up-to-date, every thread maintains its own *version map* $\nu : \mathcal{V} \to \mathbb{N}$, which associates a counter to each variable. A thread increments $\nu(x)$ whenever it writes to x. Along any execution, the version $\nu(x)$, for $x \in \mathcal{V}$, in the version map ν of thread t, associates a unique prior write with this particular valuation of x. It also reflects the total number of write accesses made (by any thread) to x to obtain the value of x stored in the map. A thread stores both its local environment and ν in the buffer after releasing a lock m. When a thread subsequently acquires lock m, it copies from the release buffers at $\mathcal{L}_m^{rel}$ the most up-to-date (according to the version numbers) value of every variable. We prove that for data race free programs, there can be only one such value. As in Section 3.2, we define *L-DRF* in terms of a labeled transition system $(\Sigma, \sigma_{ent}, TR_P)$.

States A *state* $\sigma \in \Sigma$ of the *L-DRF* semantics is a tuple $\langle pc, \mu, \Theta, \Lambda\rangle$. Here, pc and μ have the same role as in the standard semantics, i.e., they record the program counter of every thread and the ownership of locks, respectively. A versioned environment $\upsilon = \langle \phi, \nu\rangle \in VE = Env \times (\mathcal{V} \to \mathbb{N})$ is a pair comprising an environment ϕ and a version map ν. The local environment map $\Theta : \mathcal{T} \to VE$ maps every thread to its versioned environment and $\Lambda : \mathcal{L}^{rel} \to VE$ records the snapshots of versioned environments stored in buffers.

Initial State The initial state is $\sigma_{ent} = \langle \lambda t.\ ent_t, \bot, \lambda t.\ \upsilon_{ent}, \bot\rangle$, where $\upsilon_{ent} = \langle \lambda x.0, \lambda x.0\rangle$ is the initial versioned environment. In σ_{ent}, every thread is at its entry program location, no thread holds a lock, in all the versioned environments all the variables and variable versions are initialized to zero, and all the release buffers are empty.

Transition Relation The transition relation $TR_P \subseteq \Sigma \times \mathcal{T} \times \Sigma$ captures the interleaving nature of the *L-DRF* semantics of P. A transition $\tau = \langle \sigma, t, \sigma'\rangle$, also denoted by $\tau = \sigma \Rightarrow_t \sigma'$, says that thread t can execute a command which transforms state $\sigma \in \Sigma$ to state $\sigma' \in \Sigma$. We define the transition system which captures the *L-DRF* semantics of a program P by defining the transitions generated by every command in P.

Assignments and Assume Commands. We define the meaning of assignments and assume() commands (as functions from versioned environments to sets of versioned environments) by executing the standard interpretation over the environment component of a versioned environment. In addition, assignments increment the version of a variable being assigned to. Formally,

$$
\begin{aligned}
&[\![x := e]\!] : VE \to \wp(VE) = \lambda \langle \phi, \nu\rangle\ .\ \{\langle \phi[x \mapsto v], \nu[x \mapsto \nu(x) + 1]\rangle \mid v \in [\![e]\!]\phi\} \\
&[\![\texttt{assume}(b)]\!] : VE \to \wp(VE) = \lambda \langle \phi, \nu\rangle\ .\ \{\langle \phi, \nu\rangle \mid [\![b]\!]\phi\}
\end{aligned}
$$

where $[\![e]\!]\phi$, $[\![b]\!]\phi$ denote the value of the (possibly non-deterministic) expression e and the Boolean expression b, respectively, in ϕ. The set of transitions TR_c generated by an assume() or an assignment command c is given by:

$$
TR_c = \{\langle pc, \mu, \Theta, \Lambda\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu, \Theta[t \mapsto \upsilon'], \Lambda\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \upsilon' \in [\![c]\!](\Theta(t))\}
$$

Note that each thread t only accesses and modifies its *own* local versioned environment.

Acquire commands An $\texttt{acquire}(m)$ command executed by a thread t has the same effect on the lock map component in *L-DRF* as in the standard semantics. (See Section 3.2.) In addition, it updates $\Theta(t)$ based on the contents of the *relevant* release buffers. The release buffers relevant to a thread when it acquires m are the ones at $\mathcal{L}_m^{rel}$. We write $\mathcal{G}(\bar{n})$ as a synonym for $\mathcal{L}_m^{acq}$, for any post-release point $\bar{n} \in \mathcal{L}_m^{rel}$. The auxiliary function $updEnv$ is used to update the value of each $x \in \mathcal{V}$ (along with its version) in $\Theta(t)$, by taking its value from a snapshot stored at a relevant buffer which has the highest version of x, if the latter version is higher than $\Theta(t)\nu(x)$. If the version of x is highest in $\Theta(t)\nu(x)$, then t simply retains this value. Finding the most up-to-date snapshot for x (or determining that $\Theta(t)\nu(x)$ is the highest) is the role of the auxiliary function $take_x$. It takes as input $\Theta(t)$, as well as the versioned environments of the relevant release buffers, and returns the versioned environments for which the version associated with x is the highest. We separately prove that, along any execution, if there is a state in the *L-DRF* semantics σ with two component versioned environments (in thread local states or buffers) υ_1 and υ_2 such that $\upsilon_1\nu(x) = \upsilon_2\nu(x)$, then $\upsilon_1\phi(x) = \upsilon_2\phi(x)$. The set of transitions pertaining to an acquire command $c = \texttt{acquire}(m)$ is

$$TR_c = \{\langle pc, \mu, \Theta, \Lambda\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[m \mapsto t], \Theta[t \mapsto \upsilon'], \Lambda\rangle \mid \\ \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(m) = \bot \wedge \upsilon' \in updEnv(\Theta(t), \Lambda)\}$$

where $\quad updEnv : (VE \times (\mathcal{L}^{rel} \to VE)) \to \wp(VE)$ such that

$$updEnv(\upsilon, E) = \{\upsilon' \mid \textstyle\bigwedge_{x \in \mathcal{V}} \exists \upsilon_x \in take_x(Y), \upsilon'\phi(x) = \upsilon_x\phi(x) \wedge \upsilon'\nu(x) = \upsilon_x\nu(x)\}$$

with

$$Y = \{\upsilon\} \cup \{\Lambda(\bar{n}) \mid pc(t) \in \mathcal{G}(\bar{n})\}$$
$$take_x \overset{\text{def}}{=} \lambda Y \in \wp(VE).\, \{\langle\phi, \nu\rangle \in Y \mid \nu(x) = \max\{\nu'(x) \mid \langle\phi', \nu'\rangle \in Y\}\}\ .$$

For example, in Figure 1, when the program counters are $pc(t_1 \mapsto 7, t_2 \mapsto 10)$, and t_2 executes the $\texttt{acquire}()$, $take_x\ (\Theta(t_2) \cup \Lambda(7) \cup \Lambda(13)) = \Lambda(7)$. Similarly, $take_y$ also returns $\Lambda(7)$. However, $take_z$ returns $\Theta(t_2)$, since this contains the highest version of z. Thus, $updEnv\ (\Theta(t_2), \Lambda(7), \Lambda(12))$ returns the versioned environment $\langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 1^1\rangle$.

Release commands A $\texttt{release}(m)$ command executed by a thread t has the same effect on the lock map component of the state in the *L-DRF* semantics that it has in the standard semantics. (See Section 3.2.) In addition, it stores $\Theta(t)$ in the buffer associated with the post-release point pertaining to the executed $\texttt{release}(m)$ instruction. The set of transitions pertaining to a release command $c = \texttt{release}(m)$ is

$$TR_c = \{\langle pc, \mu, \Theta, \Lambda\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[m \mapsto \_], \Theta, \Lambda[n' \mapsto \Theta(t)]\rangle \mid \\ \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(m) = t\}$$

Program transition relation. The transition relation TR_P of a program P according to the *L-DRF* semantics, is the set of all possible transitions generated by its commands, and is defined as $TR_P = \bigcup_{c \in cmd(P)} TR_c$.

Collecting semantics. The collecting semantics of a program P according to the *L-DRF* semantics is the set of reachable states starting from the initial state σ_{ent}:

$$[\![P]\!] = LFP\ \lambda X.\, \{\sigma_{ent}\} \cup \{\sigma' \mid \sigma \Rightarrow_t \sigma' \in TR_P \wedge \sigma \in X \wedge t \in \mathcal{T}\}$$

4.1 Soundness and Completeness of *L-DRF* Semantics

For the class of data race free programs, the thread local semantics *L-DRF* is sound and complete with respect to the standard interleaving semantics (Section 3). To formalize the above claim, we define a function which extracts a state in the interleaving semantics from a state in the *L-DRF* semantics.

Definition 5 (Extraction Function χ).

$$\chi : \Sigma \to \mathcal{S} = \lambda \langle pc, \mu, \Theta, \Lambda \rangle . \left\langle pc, \mu, \lambda x . \Theta \left(\underset{t \in \mathcal{T}}{\operatorname{argmax}} \, \Theta(t) \, \nu(x) \right) \phi(x) \right\rangle$$

The function χ preserves the values of the program counters and the lock map, while it takes the value of every variable x from the thread which has the maximal version count for x in its local environment. χ is well-defined for *admissible* states where, if $\Theta(t)\nu(x) = \Theta(t')\nu(x)$, then $\Theta(t)\phi(x) = \Theta(t')\phi(x)$. We denote the set of admissible states by $\tilde{\Sigma}$. The *L-DRF* semantics only produces admissible states, as formalized by the following lemma:

Lemma 6. *Let $\sigma_{ent} \to_{t_1} \ldots \to_{t_N} \sigma_N$ be an execution of P in the L-DRF semantics. Then, for any σ_i, with two component versioned environments (in thread local states or buffers) υ_1 and υ_2 such that $\upsilon_1\nu(x) = \upsilon_2\nu(x)$, we have $\upsilon_1\phi(x) = \upsilon_2\phi(x)$.*

The function χ can be extended to executions in the *L-DRF* semantics by applying it to each state in the execution. The following theorems state our soundness and completeness results:

Theorem 7. ***Soundness.*** *For any trace $\pi = s_0 \to_{t_1} \ldots \to_{t_n} s_n$ of P in the standard interleaving semantics, there exists a trace $\hat{\pi} = \sigma_0 \to_{t_1} \ldots \to_{t_n} \sigma_n$ in the L-DRF semantics such that $\chi(\hat{\pi}) = \pi$. Moreover, for any transition π_i, if $c(\pi_i)$ involves a read of variable $x \in \mathcal{V}$, then $s_{i-1}\phi(x) = \sigma_{i-1}\Theta(t_i)\phi(x)$. In other words, in $\hat{\pi}$, the valuation of a variable x in the local environment of a thread t coincides with the corresponding valuation in the standard semantics only at points where t reads x.*

Theorem 8. ***Completeness.*** *For any trace $\hat{\pi}$ of P in the L-DRF semantics, $\chi(\hat{\pi})$ is a trace of the standard interleaving semantics.*

The proofs of all the claims are available in [26].

Remark 9. Till now we assumed that buffers associated with every post-release point in $\mathcal{L}_m^{rel}$ are relevant to each pre-acquire point in $\mathcal{L}_m^{acq}$, i.e., $\forall \bar{n} \in \mathcal{L}_m^{rel} : \mathcal{G}(\bar{n}) = \mathcal{L}_m^{acq}$. However, if no (standard) execution of a program contains a transition τ_i (with the target location being $\bar{n}$) which synchronizes-with a transition τ_j (with source location n), then Theorem 7 (as well as Theorem 8) holds even if we remove n from $\mathcal{G}(\bar{n})$. This is true because in race-free programs, conflicting accesses are ordered by the happens-before relation. Thus, if the most up-to-date value of a variable accessed by t was written by another thread t', then in between these accesses there must be a (sequence of) synchronization operations starting at a lock released by t' and ending at a lock acquired by t. This refinement of the set $\mathcal{G}$ based on the above observation can be used to improve the precision of the analyses derived from *L-DRF*, as it reduces the set of possible release points an acquire can observe.

5 Sequential Abstractions for Data-Race Free Programs

In this section, we show how to employ standard *sequential* analyses to compute over-approximations of the *L-DRF* semantics. Thanks to Theorem 7 and Theorem 8, the obtained results can be used to derive sound facts about the (concurrent) behavior of data race free programs in the *standard* semantics. In particular, this also allows us to establish the soundness of the *sync*-CFG analysis [9] by casting it as an abstract interpretation of the *L-DRF* semantics.

Technically, the analyses are derived by two (successive) abstraction steps: First, we abstract the *L-DRF* semantics using a thread-local cartesian abstraction which ignores version numbers and forgets the correlation between the local states of the different threads. This results in cartesian states where every program point is associated with a set of (thread-local) environments. Note that the form of these cartesian states is precisely the one obtained when computing the collecting semantics of sequential programs. Thus, they can be further abstracted using any sequential abstraction, in particular relational ones. This allows maintaining correlations between variables at all points except synchronization points (acquires and releases of locks). Note that we make the initial decision to abstract away the versions for simplicity, and we refine this abstraction later in Remark 11.

Thread-Local Cartesian Abstract Domain The abstract domain is a complete lattice over *cartesian states*, functions mapping program locations to sets of environments, ordered by pointwise inclusions. We denote the set of cartesian states by $\mathcal{A}_\times$ and range over it using $a_\times$, and define the least upper bound operator $\sqcup_\times$ in the standard way.

$$\mathcal{D}_\times \equiv \langle \mathcal{A}_\times, \sqsubseteq_\times \rangle \quad \text{where } \mathcal{A}_\times \equiv \mathcal{L} \to \wp(Env) \quad \text{and} \quad a_\times \sqsubseteq_\times a'_\times \iff \forall n \in \mathcal{L}.\, a_\times(n) \subseteq a'_\times(n)$$

The abstraction function $\alpha_\times$ maps a set of *L-DRF* states $C \subseteq \Sigma$ to a *cartesian state* $a_\times \in \mathcal{A}_\times$. The abstract value $\alpha_\times(C)(n)$ contains the collection of t's environments (where $t = tid(n))$) coming from any state $\sigma \in C$ where t is at location n. In addition, if n is a post-release point, $\alpha_\times(C)(n)$ also contains the contents of the buffer $\Lambda(n)$ for each state $\sigma \in C$. As a first cut, we abstract away the versions entirely. The concretization function $\gamma_\times$ maps a cartesian state $a_\times$ to a set of (admissible) *L-DRF* states C in which the local state of a thread t, at program point $n \in \mathcal{L}_t$, comes from $a_\times(n)$, and the contents of the release buffer pertaining to the post-release location $n \in \mathcal{L}^{rel}$ also comes from $a_\times(n)$.

$$\begin{aligned}
&\alpha_\times : \wp(\Sigma) \to \mathcal{A}_\times,\\
&\quad \text{where } \alpha_\times(C) = \lambda n \in \mathcal{L}.\, \{\phi \mid \langle pc, \mu, \Theta, \Lambda \rangle \in C \land pc(t) = n \land \langle \phi, \nu \rangle = \Theta(tid(n))\} \cup\\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\phi \mid \langle pc, \mu, \Theta, \Lambda \rangle \in C \land n \in \mathcal{L}^{rel} \land \langle \phi, \nu \rangle = \Lambda(n)\}
\end{aligned}$$

$$\begin{aligned}
&\gamma_\times : \mathcal{A}_\times \to \wp(\Sigma),\\
&\quad \text{where } \gamma_\times(a_\times) = \left\{ \langle pc, \mu, \Theta, \Lambda \rangle \in \tilde{\Sigma} \;\middle|\; \begin{array}{l} pc \in PC \land \mu \in LM \land \\ \forall t \in \mathcal{T}.\, \Theta(t) = \langle \phi, \lambda x.\_ \rangle \land \phi \in a_\times(pc(t)) \land \\ \forall n \in \mathcal{L}^{rel}.\, \Lambda(n) = \langle \phi, \lambda x.\_ \rangle \land \phi \in a_\times(n) \} \end{array} \right\}
\end{aligned}$$

Abstract Transitions The abstract cartesian semantics is defined using a transition relation, $TR^\times \subseteq \mathcal{A}_\times \times \mathcal{T} \times \mathcal{A}_\times$.

Assignments and assume commands. As we have already abstracted away the version numbers, we define the meaning of assignments and `assume()` commands c using their interpretation according to the standard semantics, denoted by $[\![c]\!]_{\mathtt{s}}$. Hence, the set of transitions coming from an `assume()` or an assignment command c is:

$$TR_c^{\times} = \left\{ a_{\times} \Rightarrow_t^{\times} a_{\times} \left[n' \mapsto a_{\times}(n') \cup \bigcup_{\phi \in a_{\times}(n)} [\![c]\!]_{\mathtt{s}}(\phi) \right] \middle| \langle n, c, n' \rangle \in inst_t \right\}$$

Acquire commands With the omission of any information pertaining to ownership of locks, an acquire command executed at program location n is only required to over-approximate the effect of updating the environment of a thread based on the contents of relevant buffers. To do so, we define an abstract *mix* operation which mixes together different environments at the granularity of single variables. The set of transitions pertaining to an acquire command $c = \mathtt{acquire}(m)$ is

$$\begin{aligned} &TR_c^{\times} = \{a_{\times} \Rightarrow_t^{\times} a_{\times}[n' \mapsto E_{mix}] \mid \langle n, c, n' \rangle \in inst_t\} \text{ , where} \\ &\quad E_{mix} = mix(a_{\times}(n') \cup \bigcup \{a_{\times}(\bar{n}) \mid n \in \mathcal{G}(\bar{n})\}) \quad \text{, and} \\ &\quad mix : \wp(Env) \to \wp(Env) \equiv \lambda B_{\times}.\{\phi' \mid \forall x \in \mathcal{V}, \exists \phi \in B_{\times} : \phi'(x) = \phi(x)\} \end{aligned}$$

In other words, the *mix* takes a cartesian product of the input states. Note that as a result of abstracting away the version numbers, a thread cannot determine the most up-to-date value of a variable, and thus conservatively picks any possible value found either in its own local environment or in a relevant release buffer.

Release commands Interestingly, the effect of release commands in the cartesian semantics is the same as `skip`: This is because the abstraction neither tracks ownership of locks nor explicitly manipulates the contents of buffers. Hence, the set of transitions pertaining to a release command $c = \mathtt{release}(m)$ is

$$TR_c^{\times} = \{a_{\times} \Rightarrow_t^{\times} a_{\times}[n' \mapsto a_{\times}(n') \cup a_{\times}(n)] \mid \langle n, c, n' \rangle \rangle \in inst_t\}$$

Collecting semantics. The collecting semantics of a program P, according to the thread-local cartesian semantics, is the cartesian state obtained as the least fixpoint of the abstract transformer obtained from $TR^{\times} = \bigcup_{c \in cmd(P)} TR_c^{\times}$ starting from $a_{\times}^{ent} = \alpha_{\times}(\{\sigma_{ent}\})$, the cartesian state corresponding to the initial state of the semantics:

$$\begin{aligned} &[\![P]\!]_{\times} = LFP\ \lambda a_{\times}.\, a_{\times}^{ent} \sqcup_{\times} \left(\bigsqcup_{\times} \{a'_{\times} \mid a_{\times} \Rightarrow_t^{\times} a'_{\times} \in TR^{\times} \wedge t \in \mathcal{T}\} \right) \text{ , where} \\ &\quad a_{\times}^{ent} = \alpha_{\times}(\{\sigma_{ent}\}) \end{aligned}$$

Theorem 10 (Soundness of Sequential Abstractions). $\gamma_{\times}([\![P]\!]_{\times}) \supseteq [\![P]\!]$.

Sequential Analyses Note that the collecting semantics of P, according to the thread-local cartesian abstraction, can be viewed as the collecting semantics of a sequential program P' obtained by adding to P's CFG edges from post-release points $\bar{n}$ to pre-acquire points n in $n \in \mathcal{G}(\bar{n})$, and where a special *mix* operator is used to combine information at the acquire points. Further, note that we abstract the environment of buffers and their corresponding release location into a single entity, which is the standard over-approximation of the set of environments at a given program location. Hence, the concurrent analysis of P can be reduced to a *sequential analysis* of P', provided a sound over-approximation of the *mix* operator is given.

Soundness of the Value-Set analysis. The analysis in [9] is obtained by abstracting the thread-local cartesian states using the value set abstraction on the environments domain. Note that in the value set domain, where every variable is associated with (an over approximation of) the set of its possible values, the mix operator reduces to a join operator.

Remark 11. We can improve upon the sequential abstraction presented earlier by not forgetting the versions entirely. We augment $\mathcal{A}_\times$ with a set S of "recency" information based on the versions as follows:

$$S = \lambda C.\{\bar{t} \mid \exists \sigma \in C, x \in \mathcal{V} : \left(\operatorname*{argmax}_{t \in \mathcal{T}} \sigma\Theta(t)\nu(x)\right) = \bar{t}\}$$

In other words, S soundly approximates the set of threads which contain the most up-to-date value of some variable $x \in \mathcal{V}$. This additional information can now be used to improve the precision of mix. We show in the experiments that the abstract domain, when equipped with this set of thread-identifiers, results in a significant gain in precision (primarily because it helps avoid spurious read-write loops between post-release and pre-acquire points, like the one in Figure 1).

6 Improved Analysis for Region Race Free Programs

In this section we introduce a refined notion of data race freedom, based on *data regions*, and derive from it a more precise abstract analysis capable of transferring *some* relational information between threads at synchronization points.

Essentially, regions are a user defined partitioning of the set of shared variables. We call each partition a *region* r, and denote the set of regions as R and the region of a variable x by $r(x)$. The semantics precisely tracks correlations between variables *within* regions *across* inter-thread communication, while abstracting away the correlations between variables across regions. With suitable abstractions, the tracked correlations can improve the precision of the analysis for programs which conform to the notion of race freedom defined below. We note that [9] and [22] do not permit relational analyses.

Region Race Freedom We define a new notion of race freedom, parameterized on the set of regions R, which we call *region race freedom* (abbreviated as *R-DRF*). *R-DRF* refines the standard notion of data race freedom by ensuring that variables residing in the same region are manipulated atomically across threads.

A *region-level data race* occurs when two concurrent threads access variables from the same region r (not necessarily the same variable), with at least one access being a write, and the accesses are devoid of any ordering constraints.

Definition 12 (Region-level races). *Let P be a program and let R be a region partitioning of P. An execution π of P, in the standard interleaving semantics, has a* region-level race *if there exists $0 \le i < j < |\pi|$, such that $c(\pi_i)$ and $c(\pi_j)$ both access variables in region $r \in R$, at least one access is a write, and it is not the case that $\pi_i \xrightarrow{hb}_\pi \pi_j$.*

Remark 13. The problem of checking for region races can be reduced to the problem of checking for dataraces as follows. We introduce a fresh variable $\mathtt{X}_r$ for each region $r \in R$. We now transform the input program P to a program P' with the following addition: We precede every assignment statement `x := e`, where r_w is the region which is written to, and $r_1, \ldots, r_n$ are the regions read, with a sequence of instructions $\mathtt{X}_{r_w} := \mathtt{X}_{r_1};\ \ldots\ \mathtt{X}_{r_w} := \mathtt{X}_{r_n};$. Statements of the form $\mathtt{assume}(b)$ do not need to be changed because b may refer only to thread-private variables. Note that these modifications do not alter the semantics of the original program (for each trace of P there is a corresponding trace in P', and vice versa). We now check for data races on the variables $\mathtt{X}_r$'s.

The *R-DRF* Semantics The *R-DRF* semantics is obtained via a simple change to the *L-DRF* semantics, a write-access to a variable x leads to incrementing the version of every variable that resides in x's region:

$$[\![x := e]\!] \colon VE \to \wp(VE) = \lambda \langle \phi, \nu \rangle \,.\, \{\langle \phi[x \mapsto v], \nu[y \mapsto \nu(y) + 1 \mid r(x) = r(y)] \rangle \mid v \in [\![e]\!]\phi\}$$

It is easy to see that Theorems 7 and 8 hold if we consider the *R-DRF* semantics instead of the *L-DRF* semantics, provided the program is region race free with respect to the given region specification. Hence, we can analyze such programs using abstractions of *R-DRF* and obtain sound results with respect to the standard interleaving semantics (Section 3).

Thread-Local Abstractions of the *R-DRF* Semantics The cartesian abstractions defined in Section 5 can be extended to accommodate regions in a natural way. The only difference lies in the definition of the *mix* operation, which now operates over *regions*, rather than variables:

$$\begin{aligned} mix : \wp(Env) \to \wp(Env) \overset{\text{def}}{=} \lambda B_\times . \{\phi' \mid \forall r \in R, \exists \phi \in B_\times : \forall x \in \mathcal{V}.\, rg(x) = r \\ \implies \phi'(x) = \phi(x)\} \end{aligned}$$

where the function rg maps a variable to its region. Mixing environments at the granularity of regions is permitted because the *R-DRF* semantics ensures that all the variables in the same region have the same version. Thus, their most up-to-date values reside in either the thread's local environment or in one of the release buffers. As before, we can obtain an effective analysis using any sequential abstraction, provided that the abstract domain supports the (more precise) region based mix operator.

7 Implementation and Evaluation

RATCOP: Relational Analysis Tool for COncurrent Programs In this section, we perform a thorough empirical evaluation of our analyses using a prototype analyzer which we have developed, called RATCOP[5], for the analysis of race-free concurrent Java programs. RATCOP comprises around 4000 lines of Java code, and implements a variety of relational analyses based on the theoretical underpinnings described in earlier sections of this paper. Through command line arguments, each analysis can be made to

[5] The project artifacts are available at `https://bitbucket.org/suvam/ratcop`

use any one of the following three numerical abstract domains provided by the Apron library [17]: Convex Polyhedra (with support for strict inequalities), Octagons and Intervals. RATCOP also makes use of the Soot [30] analysis framework. The tool reuses the code for fixed point computation and the graph data structures in the implementation of [9].

The tool takes as input a Java program with assertions marked at appropriate program points. We first checked all the programs in our benchmarks for dataraces and region races using Chord [27]. For detecting region races, we have implemented the translation scheme outlined in Remark 10 in Sec. 6. RATCOP then performs the necessary static analysis on the program until a fixpoint is reached. Subsequently, the tool automatically tries to prove the assertions using the inferred facts (which translates to checking whether the inferred fact at a program point implies the assertion condition): if it fails to prove an assertion, it dumps the corresponding inferred fact in a log file for manual inspection.

As benchmarks, we use a subset of concurrent programs from the SV-COMP 2015 suite [2]. We ported the programs to Java and introduced locks appropriately to remove races. We also use a program from [23]. While these programs are not too large, they have challenging invariants to prove, and provide a good test for the precision of the various analyses. We ran the tool in a virtual machine with 16GB RAM and 4 cores. The virtual machine, in turn, ran on a machine with 32GB RAM and a quad-core Intel i7 processor. We evaluate 5 analyses on the benchmarks, with the following abstract domains: (i) $\mathbf{A1}$**:** Without regions and thread identifiers [6]. (ii) $\mathbf{A2}$**:** With regions, but with no thread identifiers. (iii) $\mathbf{A3}$**:** Without regions, but with thread identifiers. (iv) $\mathbf{A4}$**:** With regions and thread identifiers. The analyses $\mathbf{A1}$ - $\mathbf{A4}$ all employ the Octagon numerical abstract domain. And finally, (v) $\mathbf{A5}$**:** The value-set analysis of [9], which uses the Interval domain. In terms of the precision of the abstract domains, the analyses form the following partial order: $\mathbf{A5} \prec \mathbf{A1} \prec \mathbf{A3} \prec \mathbf{A4}$ and $\mathbf{A5} \prec \mathbf{A1} \prec \mathbf{A2} \prec \mathbf{A4}$. We use $\mathbf{A5}$ as the baseline.

Porting Sequential Analyses to Concurrent Analyses. For the sequential commands, we perform a lightweight parsing of statements and simply re-use the built-in transformers of Apron. The only operator we need to define afresh is the abstract *mix*. Since Apron exposes functions to perform each of the constituent steps, implementing the abstract *mix* is straight forward as well.

Precision and Efficiency. Fig. 2 summarizes the results of the experiments.

While all the analyses failed to prove the assertions in `reorder_2`, $\mathbf{A2}$ and $\mathbf{A4}$ were able to prove them when they used convex polyhedra instead of octagons. Since none of the analyses track arrays precisely, all of them failed to prove the original assertion in `sigma` (which involves checking a property involving the sum of the array elements). However, $\mathbf{A3}$ and $\mathbf{A4}$ correctly detect a potential array out-of-bounds violation in the program. The improved precision is due to the fact that $\mathbf{A3}$ and $\mathbf{A4}$ track thread identifiers in the abstract state, which avoids spurious read-write cycles in the analysis of `sigma`. The program `twostage_3` has an actual bug, and the assertions are expected to fail. This program provides a "sanity check" of the soundness of the analyses. Programs

[6] By thread-identifiers we are referring to the abstraction of the versions outlined in Remark 11

Program	LOC	Threads	Asserts	A1		A2		A3		A4		A5	
				✓	Time (ms)	✓	Time (ms)	✓	Time (ms)	✓	Time (ms)	✓	Time (ms)
reorder_2	106	5	2	0(C)	77	2(C)	43	0(C)	71	2(C)	37	0	25
sigma ^{B}*	118	5	5	0	132	0	138	4	48	4	50	0	506
sssc12	98	3	4	4	76	4	90	4	82	4	86	2	28
unverif	82	3	2	0	115	0	121	0	84	0	86	0	46
spin2003	65	3	2	2	6	2	9	2	10	2	10	2	8
simpleLoop	74	3	2	2	56	2	61	2	57	2	64	0	27
simpleLoop5	84	4	1	0	40	0	50	0	31	0	37	0	20
doubleLock_p3	64	3	1	1	11	1	24	1	16	1	19	1	9
fib_Bench	82	3	2	0	138	0	118	0	129	0	102	0	56
fib_Bench_ Longer	82	3	2	0	95	0	103	0	123	0	91	0	35
indexer	119	2	2	2	1522	2	1637	2	1750	2	1733	2	719
twostage_3 ^{B}	93	2	2	0	61	0	48	0	57	0	28	0	59
singleton_ with_uninit	59	2	1	1	31	1	29	1	14	1	10	1	28
stack	85	2	2	0	151	0	175	0	127	0	129	0	71
stack_longer	85	1	2	0	1163	0	669	0	1082	0	1186	0	597
stack_longest	85	2	2	0	1732	0	1679	0	1873	0	2068	0	920
sync01 *	65	2	2	2	7	2	25	2	37	2	33	2	10
qw2004 *	90	2	4	0	1401	4	1890	0	1478	4	1913	0	698
[23] Fig. 3.11	89	2	2	0	49	2	46	0	54	2	36	0	19
Total	**1625**	**3 (Avg)**	**42**	**14**	**361 (Avg)**	**22**	**366 (Avg)**	**18**	**374 (Avg)**	**26**	**406 (Avg)**	**10**	**204 (Avg)**

Table 2: Summary of the experiments. Superscript ^{B} indicates that the program has an actual bug. (C) indicates the use of Convex Polyhedra as abstract data domain. * indicates a program where we have altered/weakened the original assertion.

marked with * contain assertions which we have altered completely and/or weakened. In these cases, the original assertion was either expected to fail or was too precise (possibly requiring a disjunctive domain in order to prove it). In `qw2004`, for example, we prove assertions of the form $x = y$. $\mathbf{A2}$ and $\mathbf{A4}$ perform well in this case, since we can specify a region containing x and y, which precisely track their correlation across threads. The imprecision in the remaining cases are mostly due to the program requiring *disjunctive* domains to discharge the assertions, or the presence of spurious write-write cycles which weakens the inferred facts.

Of the total 40 "valid" assertions (excluding the two in `twostage_3`), $\mathbf{A4}$ is the most precise, being able to prove 65% of them. It is followed by $\mathbf{A2}$ (55%), $\mathbf{A3}$ (45%), $\mathbf{A1}$ (35%) and, lastly, $\mathbf{A5}$ (25%). Thus, the new analyses derived from *L-DRF* and *R-DRF* perform significantly better than the value-set analysis of [9]. Moreover, this total order respects the partial ordering between the analyses defined earlier.

With respect to the running times, the maximum time taken, across all the programs, is around 2 seconds, by $\mathbf{A4}$. $\mathbf{A5}$ turns out to be the fastest in general, due to its lightweight abstract domain. $\mathbf{A2}$ and $\mathbf{A4}$ are typically slower that $\mathbf{A1}$ and $\mathbf{A3}$ respectively. The slowdown can be attributed to the additional tracking of regions by the former analyses.

Comparing with a current abstract interpretation based tool. We also compared the efficiency of RATCOP with that of Batman, a tool implementing the previous state-of-the-art analyses based on abstract interpretation [24, 25] (a discussion on the precision of our analyses against those in [24] is presented in Sec. 8). The basic structure of the benchmark programs for this experiment is as follows: each program defines a set of

#Threads	A3 Time (ms)	Bm-oct Time (ms)
2	61	7706
3	86	82545
4	138	507663
5	194	2906585
6	261	13095977
7	368	53239574

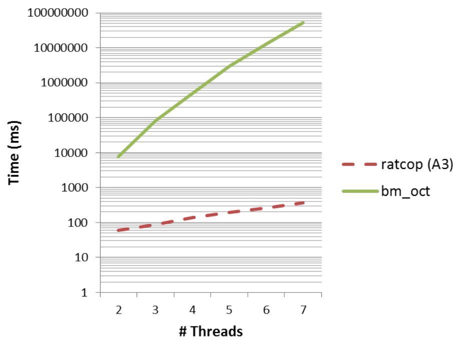


Fig. 3: Running times of RATCOP (**A3**) and Batman (Bm-oct) on loosely coupled threads. The number of shared variables is fixed at 6. The graph on the right shows the running times on a log scale.

shared variables. A `main` thread then partitions the set of shared variables, and creates threads which access and modify variables in a unique partition. Thus, the set of memory locations accessed by any two threads is disjoint. In our experiments, each thread simply performed a sequence of writes to a specific set of shared variables. In some sense, these programs represent a "best-case" scenario because there are no interferences between threads. Unlike RATCOP, the Batman tool, in its current form, only supports a small toy language and does not provide the means to automatically check assertions. Thus, for the purposes of this experiment, we only compare the time required to reach a fixpoint in the two tools. We compare **A3** against Batman running with the Octagon domain and the BddApron library [16] (Bm-oct).

The running times of the two analyses are given in Fig. 3. In the benchmarks, with increasing number of threads, RATCOP was upto 5 orders of magnitude faster than Bm-oct. The rate of increase in running time was almost linear for RATCOP, while it was almost exponential for Bm-oct. Unlike RATCOP, the analyses in [24, 25] compute sound facts at *every* program point, which contributes to the slowdown.

8 Related Work and Discussion

In this paper, we presented a framework for developing data-flow analyses for data race free shared-memory concurrent programs, with a statically fixed number of threads, and with variables having primitive data types. There is a rich literature on concurrent dataflow analyses and [28] provides a detailed survey of some of them. We compare some of the relevant ones in this section. [5] automatically lifts a given sequential analysis to a sound analysis for concurrent programs, using a datarace detector. Here, dataflow facts are not communicated across threads, and this can lose a lot of precision. The work in [4, 22] allows a greater degree of inter-thread communication. However, unlike our semantics, they are unable to infer relational properties between variables. The methods described in [9, 10, 15] present concurrent dataflow algorithms by building specialized concurrent flow graphs. However, the class of analyses they address are restricted – [10] handles properties expressible as Quantified Regular Expressions, [15] handles reaching definitions, while [9] only handles value-set analyses.

```
1  acquire(m)
2  x := 1
3  y := 1
4  release(m)
```

(a) Thread 1

```
6  while p ≠ 1 do {
7    acquire(m)
8    p := y
9    release(m)
10 }
11 x := 2
12 p := x
13 assert (p ≠ 1)
```

(b) Thread 2

Fig. 4: Example demonstrating that a program can be DRF, even when a read from a global variable is not directly guarded by any lock.

In [24], an abstract interpretation formulation of the rely-guarantee proof technique [18, 31] is presented in the form of a precise semantics. The semantics in [24] involves a nested fixed-point computation, compared to our single fixed-point formulation. The analysis aims to be sound at *all* program points (e.g, in Fig. 1 the value of y at line 9 in t_2), due to which many more interferences will have to be propagated than we do, leading to a less efficient analysis. Moreover, for certain programs, our abstract analyses are more precise. Fig. 4 shows a program which is race free, even though the conflicting accesses to x in lines 2 and 12 are not protected by a common lock. The "lock invariants" in [24] would consider these accesses as potentially racy, and would allow the read at line 12 to observe the write at line 2, thereby being unable to prove the assertion. However, our analyses would ensure that the read only observes the write at line 11, and is able to prove the assertion. [13] presents an operational semantics for concurrent programs, parameterized by a relation. It makes additional assumptions about code regions which are unsynchronized (allowing only read-only shared variables and local variables in such regions). Moreover, it too computes sound facts at every point, resulting in less efficient abstractions.

A traditional approach to analyzing concurrent programs involves *resource invariants* associated with every lock (e.g. [14]). This approach depends on a *locking policy* where a thread only accesses global data if it holds a protecting lock. In contrast, our approach does not require a particular locking policy (e.g., see Fig. 4), and is based on a parameterized notion of data-race-freedom, which allows to encode locking policies as a particular case. Thus, our new semantics provides greater flexibility to analysis writers, at the cost of assuming data-race-freedom.

Our notion of region races is inspired by the notion of high-level data races [1]. The concept of splitting the state space into regions was earlier used in [21], which used these regions to perform shape analysis for concurrent programs. However, that algorithm still performs a full interleaving analysis which results in poor scalability. The notion of variable packing [3] is similar to our notion of data regions. However, variable packs constitute a purely *syntactic* grouping of variables, while regions are semantic in nature. A syntactic block may not access all variables in a semantic region, which would result in a region partitioning more refined than what the programmer has in mind, which would result in decreased precision. In contrast to our approach, the techniques in [11, 12] provide an approach to verifying properties of concurrent programs using *data flow* graphs, rather than use control flow graphs like we do.

As future work, we would like to evaluate the performance of our tool when equipped with disjunctive relational domains. In this paper, we do not consider dynamically allocated memory, and extending the *L-DRF* semantics to account for the heap memory is interesting future work. Abstractions of such a semantics could potentially yield efficient shape analyses for race free concurrent programs.

Acknowledgments. We would like to thank the anonymous reviewers for their insightful and helpful comments. This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° [321174], by Len Blavatnik and the Blavatnik Family foundation, and by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University.

References

1. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *New Technologies for Information Systems, Proceedings of the 3rd International Workshop on New Developments in Digital Libraries, NDDL 2003*, pages 82–93, 2003.
2. Dirk Beyer. Software verification and verifiable witnesses. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.
3. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *CoRR*, abs/cs/0701193, 2007.
4. Jean-Loup Carre and Charles Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. *CoRR*, abs/0910.5833, 2009.
5. Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 316–326, 2008.
6. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
8. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 84–96. ACM, 1978.
9. Arnab De, Deepak D'Souza, and Rupesh Nasre. Dataflow analysis for datarace-free programs. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011*, pages 196–215, 2011.
10. Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 6-9, 1994*, pages 62–75, 1994.
11. Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–308, 2012.

12. Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 129–142, 2013.
13. Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized memory models and concurrent separation logic. In *European Symposium on Programming*, pages 267–286. Springer, 2010.
14. Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 266–277, 2007.
15. Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 159–168, 1993.
16. Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electronic Notes in Theoretical Computer Science*, 267(2):29–42, 2010.
17. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009.
18. Cliff B. Jones. *Developing methods for computer programs including a notion of interference*. PhD thesis, University of Oxford, UK, 1981.
19. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.
20. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979.
21. Roman Manevich, Tal Lev-Ami, Mooly Sagiv, Ganesan Ramalingam, and Josh Berdine. Heap decomposition for concurrent shape analysis. In *Static Analysis, 15th International Symposium, SAS*, pages 363–377, 2008.
22. Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012.
23. Antoine Miné. *Static analysis by abstract interpretation of concurrent programs*. PhD thesis, Ecole Normale Supérieure de Paris-ENS Paris, 2013.
24. Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation*, pages 39–58. Springer, 2014.
25. Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–404. Springer, 2017.
26. Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. Thread-Local Semantics and its Efficient Sequential Abstractions for Race-Free Programs. `http://www.csa.iisc.ernet.in/TR/2016/3/sasTechReport.pdf`.
27. Mayur Naik. Chord: A Program Analysis Platform for Java. `http://www.cis.upenn.edu/~mhnaik/chord.html`. Accessed: 2017-03-27.
28. Martin C. Rinard. Analysis of multithreaded programs. In *Static Analysis, 8th International Symposium, SAS*, pages 1–19, 2001.
29. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP*, pages 27–37, 1997.
30. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.
31. Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9:149–174, 1997.