RATCOP: Relational Analysis Tool for Concurrent 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. In this paper, we present RATCOP, a static analysis tool for efficiently computing relational invariants in race free shared-variable multi-threaded Java programs. The tool trades the standard sound-at-all-program-points guarantee for gains in efficiency. Instead, it computes sound facts for a variable only at program points where it is "relevant". In our experiments, RATCOP was fairly precise while being fast. As a tool, RATCOP is easy-to-use, and easily extensible.

1 Introduction

Writing efficient and correct multi-threaded programs is an onerous task, since a multi-threaded program admits a large set of possible behaviors. As a result, such programs provide fertile ground for many insidious defects: the bugs are difficult to detect, difficult to reproduce, and can result in unpredictable failures. Thus, developers are greatly aided by tools which can automatically report such defects.

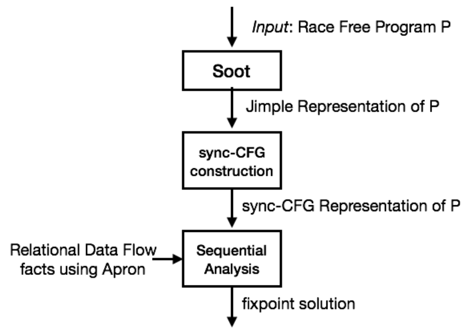
Unfortunately, designing algorithms which can automatically reason about behaviors of concurrent programs is also a very hard problem. Key to the difficulty lies in accounting for the large set of inter-thread interactions. Static analysis algorithms, based on the abstract interpretation framework [3], compute sound approximations of the set of "concrete states" arising at each program point. With this notion of soundness, a precise static analyzer does not usually scale, whereas a fast analysis is usually quite imprecise [2].

In this paper, we describe RATCOP [3]: **R**elational **A**nalysis **T**ool for **CO**ncurrent **P**rograms, a tool to efficiently compute relational invariants in shared-memory data race free multi-threaded Java programs. RATCOP does not handle procedure calls or dynamic memory allocation. The abstract analyses implemented in RATCOP are based on a novel *thread-local* semantics, called *L-DRF* [7]. Here, each thread maintains a local copy of the global state. When a thread t executes a non-synchronization command (an assignment or an `assume`), it operates on its local state alone. Each `release` instruction is associated with a "buffer". When t executes a `release(m)` command, it stores a copy of its local state in the corresponding buffer. When a thread t' subsequently acquires `m`, it is allowed to observe the states stored at a set of "relevant" buffers. t' then performs a mix of these states to create a fresh local state. As [7] shows, for data race free (DRF) programs, each trace in the standard semantics corresponds to some trace in the *L-DRF*

[3] The source code of RATCOP is available at `https://bitbucket.org/suvam/ratcop`

semantics, and vice versa. Thus, the *L-DRF* semantics is a precise description of the behaviors of DRF programs.

The *L-DRF* semantics allows one to rapidly port existing sequential analyses to analyses for race free programs. Such analyses operate on a program graph called sync-CFG (first introduced in [4]), which is a collection of the control-flow graphs of each thread, augmented with synchronization edges between the release of a lock `m`, and an acquire of `m`. Consequently, the sync-CFG restricts inter-thread propagations to synchronization points alone. The resulting analyses satisfy a non-standard notion of soundness: the computed facts for a variable are sound only at program points where it is *accessed*. A more precise analysis is obtained by parameterizing *L-DRF* with a user-defined partitioning of the program variables, resulting in a semantics called *R-DRF*. Each partition is also called a "region". Assuming that the input program is free from *region races* [7], which is a stronger notion than data races, the resulting abstract analyses are more precise than those derived from *L-DRF*.



Input: Race Free Program P
Soot
Jimple Representation of P
sync-CFG construction
sync-CFG Representation of P
Relational Data Flow facts using Apron
Sequential Analysis
fixpoint solution

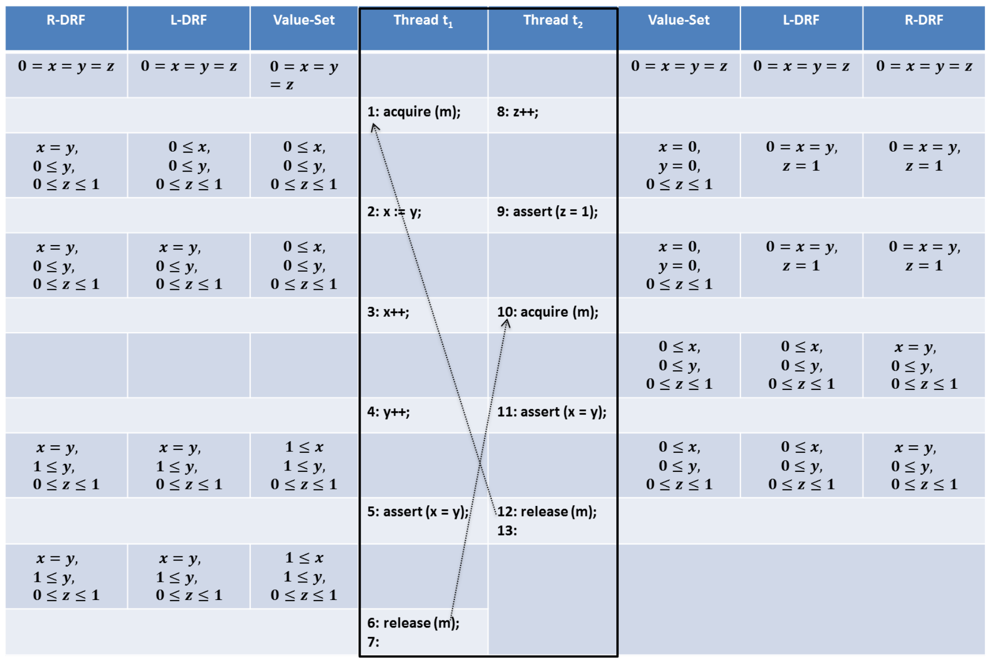
Fig. 1. High-level overview of RATCOP

In RATCOP, we instantiate abstractions of *L-DRF* and *R-DRF* to create several relational analyses with varying degrees of precision. Our objective was two-fold: (i.) to investigate the ease of porting a sequential relational analysis to an analysis for race free concurrent programs (ii.) to investigate the efficiency and precision of the resulting analysis. The base-line is an interval analysis derived from an earlier work [4]. RATCOP makes use of the Soot [8] and Apron [5] libraries. RATCOP intelligently leverages the race freedom property of the input program to minimize the number of inter-thread data flow propagations, while retaining a fair degree of precision. As shown in [7], on the benchmarks, RATCOP was able to prove upto 65% of the assertions, in comparison to 25% achieved by the base-line analysis. On a separate set of benchmarks, RATCOP was upto 5 orders of magnitude faster than Batman, a recent static analyzer for concurrent programs [6]. Finally, RATCOP is easy-to-use, quite robust, and easily extensible. In this paper, we detail the architecture of RATCOP.

2 Architecture of RATCOP

RATCOP comprises around 4000 lines of Java code, and implements a number of relational analyses with varying degrees of precision and scalability. Through command line arguments, the tool can make use of the following three abstract domains provided by Apron: convex polyhedra, octagons and intervals. It takes only a few lines of code to extend RATCOP to use additional numerical abstract domains.

RATCOP assumes that the input program is free from data races, and does not perform any explicit checks for the same. To detect region-level races, RATCOP implements the scheme outlined in [7], which reduces the problem of checking for region-level races to that of checking for data races on specific "auxiliary" variables.



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. 2. An example from [7] illustrating the relational analyses implemented in RATCOP. The *sync*-CFG representation of the program is given at the center: *inter*-thread communication is restricted to synchronization points alone. All the variables are shared and initialized to 0. The Value-Set column shows the facts computed using an interval analysis derived from [4]. The *L-DRF* and *R-DRF* columns show the facts computed by polyhedral abstractions of the thread-local semantics, and its region-parameterized version. The *R-DRF* analysis is able to prove all the 3 assertions, the *L-DRF* proves 2, while the Value-Set analysis only proves 1 assertion.

RATCOP re-uses the code to construct the *sync*-CFG representation of a program from the implementation of [4]. The *sync*-CFG construction makes use of a pointer-analysis, coupled with a may-happens-in-parallel analysis.

The tool now performs a *sequential* analysis, with the only additional operator being the inter-thread join. Once the fixpoint is reached, RATCOP automatically tries to prove the assertions in the program, which amounts to checking whether the computed facts at a program point imply the condition being asserted. If the tool fails to prove the implications, the assertion condition and the corresponding data flow fact is logged for further manual investigation.

For the non-synchronization instructions, RATCOP performs some light parsing, followed by re-using the existing sequential transformers exposed by Apron. The only operator we define afresh is the *inter*-thread join, which is used at the `acquire` points. However, this turns out to be simple as well, being a combination of operations provided by Apron. Thus, porting a sequential relational analysis based on Apron to an analysis for a race free concurrent program, using our framework, turns out to be quite straightforward. Fig. 1 summarizes the set of operations in RATCOP.

RATCOP implements 5 relational analyses: $\mathbf{A1} - \mathbf{A4}$, are derived from the *L-DRF* and *R-DRF* semantics, and use the octagon domain. The fifth, $\mathbf{A5}$ (which is also our baseline), is an interval analysis derived from [4]. The analyses differ in the degree of abstraction from the *L-DRF* and *R-DRF* semantics, with $\mathbf{A4}$ using the most precise abstract domain, and $\mathbf{A5}$ being the least.

3 Experiments

We illustrate the operation of RATCOP on a simple program from [7], shown in Fig. 2. The program is free from data races. If the regions are defined to be $\langle\{x, y\}, \{z\}\rangle$, then the program is free from region races as well [4]. The results of $\mathbf{A5}$ are shown under the column "Value-Set". Since an interval based analysis is the best we can do using [4], the resulting analysis is quite imprecise: it is only able to prove the assertion at line 9. The analysis cannot track any relational properties. We do better with $\mathbf{A2}$, derived from *L-DRF*, which uses octagons. This analysis *does* track the correlation between x and y, which allows it to be additionally prove the assertion at line 5. However, the inter-thread mixing (at the `acquire` points) is done at the granularity of individual variables. This keeps $\mathbf{A2}$ from inferring $x = y$ at line 12, for example, even though the two incoming edges clearly maintain this invariant. The analysis $\mathbf{A4}$ performs this mixing at the granularity of the specified regions. Thus, it is able to prove all 3 assertions.

In our experiments in [7], we used a subset of concurrent programs from the SV-COMP 2015 suite [1], after porting them to Java and introducing locks appropriately to remove races. We ran our experiments in a virtual machine with 16GB RAM and 4 cores which, in turn, ran on a machine with 32 RAM and a quad-core Intel i7 processor. Unsurprisingly, $\mathbf{A4}$ was the most precise, being able to prove 65% of the assertions. It was also the slowest, the average time being 406ms. $\mathbf{A5}$ was the least precise, having proved 25% of the assertions with an average time of 204ms.

We compared RATCOP with a current abstract interpretation based tool for multi-threaded programs [6], called Batman. Unlike RATCOP, which handles a large subset of multi-threaded Java programs, Batman handles a toy language with limited constructs. Moreover, Batman does not automatically check the validity of assertions, which renders it difficult to use with even small programs. We evaluated the two tools on multi-threaded programs with little inter-thread communication. RATCOP leveraged the lack of inter-thread communication intelligently to perform up to 5 orders of magnitude faster than Batman. The key difference between the two tools is that Batman tries to compute sound facts at *every* program point, whereas RATCOP computes sound facts for variables only at program points where they are *accessed*.

4 Conclusion

In this paper, we presented RATCOP: a static analysis tool which efficiently computes relational invariants for race free concurrent programs, with a non-standard notion of soundness. We hope that RATCOP will serve as a stepping stone for future static analyses for the class of race free programs.

[4] The interested reader may refer to [7] for the exact definition of region races.

References

1. 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.
2. 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*, pages 316–326, 2008.
3. 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.
4. 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.
5. 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.
6. 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.
7. Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. Thread-local semantics and its efficient sequential abstractions for race-free programs. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 253–276, 2017.
8. 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.