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`

O. Strichman and R. Tzoref-Brill (Eds.): HVC 2017, LNCS 10629, pp. 229–233, 2017.
https://doi.org/10.1007/978-3-319-70389-3_18

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*.

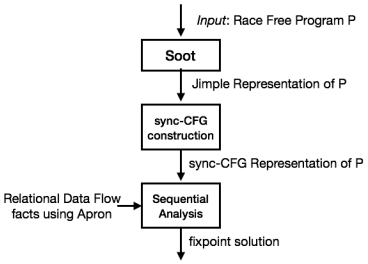

Fig. 1. High-level overview of RATCOP

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.