Information and Computation ••• (••••) •••–•••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Lost in abstraction: Monotonicity in multi-threaded programs ☆

Alexander Kaiser [a], Daniel Kroening [a], Thomas Wahl [b,*]

[a] University of Oxford, United Kingdom
[b] Northeastern University, Boston, United States

ARTICLE INFO

Article history:
Received 15 December 2014
Available online xxxx

Keywords:
Multi-threaded software
Parameterized verification
Monotonicity
Predicate abstraction

ABSTRACT

Monotonicity in concurrent systems stipulates that, in any global state, system actions remain executable when new processes are added to the state. This concept is both natural and useful: if every thread's memory is finite, monotonicity often guarantees the decidability of safety properties even when the number of running threads is unknown. In this paper, we show that finite-data thread abstractions for model checking can be at odds with monotonicity: predicate-abstracting monotone software can result in non-monotone Boolean programs — the monotonicity is *lost in the abstraction*. As a result, pertinent well-established safety checking algorithms for infinite-state systems become inapplicable. We demonstrate how monotonicity in the abstraction can be restored, without affecting safety properties. This improves earlier approaches of enforcing monotonicity via overapproximations. We implemented our solution in the unbounded-thread model checker monabs and applied it to numerous concurrent programs and algorithms, whose predicate abstractions are often fundamentally beyond existing tools.

1. Introduction

Multi-threading is becoming a premier technology for accelerating computations in a post frequency-scaling era. The widespread availability of thread libraries for mainstream languages including C and Java, as well as for all major operating systems, makes this technology easily accessible. This can entrap the inexperienced programmer to create code with puzzling and irreproducible behavior. The software community needs to respond to this threat in part by providing formal technology that exposes potential bugs in concurrent programs before deployment.

Towards this end, this paper proposes a safety analysis method for non-recursive procedures executed by multiple threads (e.g. dynamically generated, and possibly unbounded in number), which communicate via shared variables and higher-level mechanisms such as mutexes. OS-level code, including Windows, UNIX, and Mac OS device drivers, makes frequent use of such concurrency APIs, whose correct use is therefore critical to ensure a reliable programming environment.

The verification method we propose is based on *predicate abstraction*. The utility of this technique is known to depend critically on the choice of predicates: the consequences of a poor choice range from inferior performance to flat-out unprovability of certain properties. We introduce an extension of predicate abstraction to multi-threaded programs that enables reasoning about intricate data relationships, namely

☆ Support provided by the Toyota Motor Corporation, NSF grant 1253331, and ERC project 280053.
* Corresponding author.

http://dx.doi.org/10.1016/j.ic.2016.03.003
0890-5401/

```
struct Spinlock {
      natural s := 1;      // ticket being served
      natural t := 1; };   // next free ticket

struct Spinlock lock;                    // shared

void spin_lock() {
      natural l := 0;                    // local
  ℓ1: l := fetch_and_add(lock.t);
  ℓ2: while (l ≠ lock.s)
        /* spin */; }

void spin_unlock() {
  ℓ3: lock.s++; }
```

The ticket algorithm: Shared variable `lock` has two natural-number components: `s` holds the number of the ticket currently being served, while `t` holds the next free number (if no thread is currently served, we have `s = t`).
To request access to the protected region, a thread atomically retrieves the value of `t` into local variable `l` and then increments `t`. The thread then spins until `l` equals `s`. To unlock, `s` is incremented.

Fig. 1. The ticket algorithm — Our goal is to verify standard *parameterized mutual exclusion*: no matter how many threads try to acquire and release the lock concurrently, no two of them are simultaneously between the calls to functions spin_lock and spin_unlock.

shared-variable: "shared variables `s` and `t` are equal",
single-thread: "local variable `l` of thread *i* is less than shared variable `s`", and
inter-thread: "local variable `l` of thread *i* is less than variable `l` *in all other threads*".

Why such a rich predicate language? For certain concurrent algorithms such as the widely used *ticket* busy-wait lock algorithm [1] (the default locking mechanism in the Linux kernel since 2008; see Fig. 1), the verification of elementary safety properties *requires* inter-thread relationships (see Sect. 2.2). They are needed to express, for instance, that a thread holds the minimum ticket value, an inter-thread relationship.

In the main part of the paper, we address the problem of full parameterized (unbounded-thread) program verification with respect to our predicate language. Such reasoning requires first that the *n*-thread abstract program $\hat{\mathcal{P}}^n$, obtained by existential predicate abstraction of the *n*-thread concrete program $\mathcal{P}^n$, is rewritten into a generic template program $\tilde{\mathcal{P}}$ to be executed by (any number of) multiple threads. In order to capture the semantics of these programs in the template $\tilde{\mathcal{P}}$, the template programming language, too, must permit variables that refer to the currently executing thread, or to *all* passive (non-executing) threads; we call such programs *dual-reference (DR)*. We describe how to obtain $\tilde{\mathcal{P}}$, namely as an overapproximation of $\hat{\mathcal{P}}^{\mathsf{b}}$, for a constant b that grows linearly with the number of inter-thread predicates used in the abstraction.

Given a *Boolean* dual-reference program $\tilde{\mathcal{P}}$ (obtained from predicate abstraction), we might expect the unbounded-thread replicated program $\tilde{\mathcal{P}}^{\infty}$ to form a classical *well quasi-ordered transition system* [2], enabling the fully automated, algorithmic safety property verification in the abstract. This turns out not to be the case: the expressiveness of dual-reference programs renders parameterized program location reachability undecidable, despite the finite-domain variables. The root cause is the lack of *monotonicity* of the transition relation with respect to the standard partial order over the space of unbounded thread counters. That is, adding passive threads to the source state of a valid transition can invalidate this transition. Since the input C programs are, by contrast, typically monotone, we say the monotonicity is *lost in the abstraction*. As a result, our abstract programs are in fact not well quasi-ordered.

Inspired by earlier work on *monotonic abstractions* [3], we address this problem by restoring the monotonicity using a simple *closure operator*, which enriches the transition relation of the abstract program $\tilde{\mathcal{P}}$ such that the obtained program $\tilde{\mathcal{P}}_m$ gives rise to a monotone (and thus well quasi-ordered) system. The closure operator essentially terminates passive threads that block transitions allowed by other passive threads. In contrast to earlier work [3], which enforces monotonicity in genuinely *non*-monotone systems, we exploit the monotonicity of the input programs. As a result, the monotone closure $\tilde{\mathcal{P}}_m$ can be shown to be safety-equivalent to the non-monotone program $\tilde{\mathcal{P}}$.

To summarize, the core contribution of this paper is a predicate abstraction strategy for asynchronous unbounded-thread C programs, with respect to the rich language of inter-thread predicates. This language allows the abstraction to track properties that are essentially universally quantified over all passive threads. We have implemented our technique in the infinite-state model checker monabs. We observe that our tool is able to verify certain parameterized programs (such as the ticket algorithm) that are fundamentally beyond existing tools we are aware of [4–11]. The reasons vary from their inability to deal with unbounded threads, to lacking support for inter-thread predicates. We include an extensive experimental evaluation on system-level concurrent software and synchronization algorithms.

2. Inter-thread predicate abstraction

In this section we introduce single- and inter-thread predicates, with respect to which we then formalize existential predicate abstraction. Except for the extended predicate language, these concepts are mostly standard and lay the technical foundations for the contributions of this paper.

Please cite this article in press as: A. Kaiser et al., Lost in abstraction: Monotonicity in multi-threaded programs, Inf. Comput. (2016), http://dx.doi.org/10.1016/j.ic.2016.03.003