2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

# Poster: Static Analysis of Concurrent Higher-Order Programs

Quentin Stievenart, Jens Nicolay, Wolfgang De Meuter, Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be

***Abstract*—Few static analyses support concurrent higher-order programs. Tools for detecting concurrency bugs such as deadlocks and race conditions are nonetheless invaluable to developers. Concurrency can be implemented using a variety of models, each supported by different synchronization primitives. Using this poster, we present an approach for analyzing concurrent higher-order programs in a precise manner through abstract interpretation. We instantiate the approach for two static analyses that are capable of detecting deadlocks and race conditions in programs that rely either on compare-and-swap (`cas`), or on conventional locks for synchronization. We observe few false positives and false negatives on a corpus of small concurrent programs, with better results for the lock-based analyses. We also observe that these programs lead to a smaller state space to be explored by the analyses. Our results show that the choice of synchronization primitives supported by an abstract interpreter has an important impact on the complexity of the static analyses performed with this abstract interpreter.**

## I. Introduction

It is difficult to find concurrency bugs such as deadlocks and race conditions by hand. An exponential number of thread interleavings needs to be explored. Higher-order programming features, such as functions taking others as arguments, exacerbate this difficulty. The need for tool support increases as these features become more widespread.

The AAM approach to abstract interpretation [9] provides a theoretical foundation for this support. It has already been adapted to concurrent programs [6], giving rise to the P(CEK$^\star$)S machine that features support for multiple threads of execution (i.e., through primitives `spawn` and `join`), as well as for their synchronization through a compare-and-swap primitive (i.e., `cas`). In addition to supporting higher-order features, its qualities include conciseness and extensibility.

We extend the P(CEK$^\star$)S machine with first-class support for locks as a synchronization primitive and demonstrate its applicability as the foundation for two client analyses that detect deadlocks and race conditions [8]. The corresponding implementations are publicly available[1].

## II. Foundation

We observed that the P(CEK$^\star$)S machine only scales to small concurrent programs (around 60 lines of code with 3 threads). This is due to the state explosion problem, also shared by model checking techniques that support concurrent programs. Since every thread interleaving is computed, exploring the corresponding state space quickly becomes intractable. Research on this problem in the area of model checking (cf. [4]) has lead to techniques such as partial-order reduction [7], binary decision diagrams [3], and bounded model checking [1]. However, we leave incorporating these techniques in the P(CEK$^\star$)S machine as future work.

Instead, we orthogonally address another source of complexity in the generated state space: the synchronization primitives supported by the abstract interpreter. The original P(CEK$^\star$)S machine uses an atomic compare-and-swap synchronization primitive, `cas`. This primitive works in a *active* way: `(cas v old new)` will compare the content of the variable `v` with the value `old`, and only if the content matches, `v` will be updated to take the value of `new`. If the comparison fails (e.g., due to a bad thread interleaving, which would make the computation of `new` incorrect if based on the value `old`), the update has to be tried again. This is how a shared concurrent counter would be implemented using `cas`:

```
(letrec ((v 0)
         (inc (lambda ()
                (let ((old v) (new (+ old 1)))
                  (if (cas v old new)
                    #t ;; successful update
                    ;; incorrect update (v != old),
                    ;; try again
                    (inc))))))
  [...])
```

The fact that a failing `cas` has to be retried over and over until it eventually succeeds, leads to an increase in the size of the state space where many states are generated only because a `cas` is retried, and will not lead to real progress in the program execution.

Another common synchronization mechanism is locking. Locks can be simulated with `cas`, and no first-class language support is necessary to use locks. The lock-based equivalent of the shared concurrent counter example is shown below.

Based on the observations that `cas` introduces an overhead in terms of number of states, and that most programs tend to use locks instead of `cas`, we add first-class support for locks to the P(CEK$^\star$)S machine. This way, the abstract interpreter no longer needs to analyze a low-level library implementation of locking in terms of `cas`. A lock can be either `#locked` or `#unlocked`. It can be acquired with the `acquire` primitive,

[1] https://github.com/acieroid/pcesk

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.265

```
(letrec ((v 0)
         (a-lock #unlocked)
         (inc (lambda ()
                (acquire a-lock)
                (set! v (+ v 1))
                (release a-lock))))
  [...])
```

which blocks the current thread until the lock is acquired. Once acquired, a lock can be released with the `release` primitive.

## III. Client Analyses

One can design static analyses for concurrent programs as an exploration of the state space generated by the P(CEK$^\star$)S machine. We have formulated analyses for detecting deadlocks and race conditions, each for programs using `cas` and for programs using first-class locks [8]. We have observed that analyses for programs making use of locks are more straightforward to formulate. For example, the race condition analysis with locks is equivalent to a conflict analysis. With `cas`, the conflict analysis has to be augmented by another analysis to catch a source of race conditions not present with locks.

## IV. Evaluation

We tested our client analyses on a small number of concurrent programs exhibiting various situations of deadlocks and race conditions. The results are given in Table I. Lock-based analyses tend to be able to handle programs with more threads and locks, and have fewer false positives when dealing with deadlocks. The only false positive found by both race condition analyses is a benign race condition which has no effect on the outcome of the programs, and it is arguable whether this instance of race condition should be detected.

TABLE I
Results of the deadlock and race condition analyses.

| | Deadlock Analysis | | Race Condition Analysis | |
|---|---|---|---|---|
| | `cas` | locks | `cas` | locks |
| Input programs | 6 | 7 | 8 | 6 |
| Max. LOC | 65 | 58 | 45 | 52 |
| Max. threads | 3 | 4 | 3 | 4 |
| Max. locks | 2 | 3 | – | 1 |
| Max. states | 234962 | 3829 | 8845 | 3829 |
| Defects | 7 | 5 | 7 | 2 |
| Found | 6 | 5 | 6 | 2 |
| False positives | 2 | 0 | 1 | 1 |

Since locks are a blocking synchronization technique, programs using locks generate smaller state spaces. Indeed, a thread waiting to acquire a lock will be blocked, and will not introduce any new states until it can acquire the lock, whereas it would have introduced new states if `cas` were used. We verify that this is actually the case by measuring the state space size of similar programs implemented once using `cas`, and once using locks. We identify that the switch from `cas` to locks leads to around one order of magnitude improvement, and up to three orders of magnitude improvement in the number of states. This improvement allows the analysis to support more complex programs, involving more threads.

## V. Conclusion

Our poster presents the P(CEK$^\star$)S machine and its design based on the CESK machine. The different synchronization mechanisms supported (`cas` and locks) are introduced through examples. We show that switching from `cas` to first-class locks in the analyzed programs can lead to a reduction of the state space of around one order of magnitude, and up to three orders of magnitude. We also introduce our deadlock and race condition analyses.

This approach of performing static analysis for concurrent programs has the advantage of supporting higher-order programs, unlike most existing concurrent static analyses. It also outputs a precise state graph that over-approximates executions of the program. One downside of this approach is the size of the state graph (up to $10^6$ states in our examples), which is subject to the state explosion problem, as concurrent model checkers are. However, it allows a relatively straightforward formulation of static analyses, such as our deadlock and race condition analyses.

Future work includes tackling the state explosion problem to improve the analysis' scalability to support programs of a complexity closer to real-world applications. We aim to reduce the size of the state space further, by adapting techniques that have been proven useful in the context of model checking (e.g., to verify models of up to $2^{120}$ states [2]). One such technique is cartesian partial-order reduction [5], which is formulated in a manner close to abstract interpretation.

Adapting the P(CEK$^\star$)S machine to more precise abstract interpretation models (e.g., pushdown-based models) could improve the precision of the derived states, but requires a non straightforward adaptation. Finally, adapting the P(CEK$^\star$)S machine to other concurrency models and other synchronization primitives could lead to further improvements and new insights.

## References


[1] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[2] Jerry Burch, Edmund M Clarke, and David Long. Symbolic model checking with partitioned transition relations. *Computer Science Department*, page 435, 1991.

[3] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439. IEEE, 1990.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[5] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. *Cartesian partial-order reduction*. Springer, 2007.

[6] Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis*, pages 180–197. Springer, 2011.

[7] Doron Peled. Ten years of partial order reduction. In *Computer Aided Verification*, pages 17–28. Springer, 1998.

[8] Quentin Stievenart. Static analysis of concurrent constructs in higher-order programs. Master's thesis, Université Libre de Bruxelles, Belgium, 2014.

[9] David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.