Detecting Concurrency Bugs in Higher-Order Programs through Abstract Interpretation

Quentin Stievenart
Vrije Universiteit Brussel, Belgium
qstieven@vub.ac.be

Jens Nicolay
Vrije Universiteit Brussel, Belgium
jnicolay@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel, Belgium
wdmeuter@vub.ac.be

Coen De Roover
Vrije Universiteit Brussel, Belgium
cderoove@vub.ac.be

Abstract

Manually detecting bugs in concurrent programs is hard due to the myriad of thread interleavings that needs to be accounted for. Higher-order programming features only exacerbate this difficulty. The need for tool support therefore increases as these features become more widespread. We investigate the P(CEK*)S abstract machine as the foundation for tool support for detecting concurrency bugs. This abstract interpreter analyzes multi-threaded, higher-order programs with shared-store concurrency and a compare-and-swap synchronization primitive. In this paper, we evaluate two different approaches to reduce the size of the state space explored by the abstract interpreter. First, we integrate abstract garbage collection into the abstract interpreter, and we observe that it does not reduce the state space as expected. We then evaluate the impact of adding first-class support for locks on the machine's client analyses. To this end, we compare a `cas`-based and a lock-based formulation of race condition and deadlock detection analyses. We show that adding first-class support for locks not only significantly reduces the number of abstract program states that need to be explored, but also simplifies formulating the client analyses.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.1.3 [*Software*]: Concurrent Programming

General Terms Theory, Languages, Verification

Keywords Abstract Interpretation, Deadlock Detection, Race Condition Detection

1. Introduction

Concurrent programs can contain errors that are particularly hard to find, such as race conditions and deadlocks. These tend to arise only with certain thread interleavings and in a non-deterministic manner. Tool support is in order, but it has proven difficult to provide this for higher-order languages. In higher-order languages, functions are first-class values that are allowed to flow freely through the program in the same way as other values such as integers and strings. Because function calls influence control flow, this results in a mutual dependency between control flow and value flow.

In this work, we advance towards static tool support for detecting concurrency bugs in higher-order programs with side-effects. Our goal is to provide this support with reasonable accuracy, and without requiring input from users. The P(CEK*)S machine, described by Might and Van Horn [27], provides a suitable foundation for this support. They adapt a sequential CESK machine, modeling the semantics of a single thread, into the P(CEK*)S machine that allows for concurrent threads of execution. The result is a modular semantics, in which the concurrency concerns are separated from sequential semantics. The original formulation of the P(CEK*)S machine adds three concurrency constructs: `spawn` to create new threads, the blocking construct `join` to wait for a thread to complete, and `cas` (compare-and-swap) for synchronizing threads.

In this paper, we make the following contributions:

- We transpose abstract garbage collection [26] from the sequential CESK machine it was originally proposed for, to the P(CEK*)S machine. In our experiments, the expected reductions in state space resulting from this optimization technique are less pronounced than in the CESK setting. We observe no consistent improvements in precision and performance as a result.
- We extend the original P(CEK*)S machine with first-class support for locks. Before, it was up to application developers to implement them in terms of `cas`. We experimentally show that this addition yields an important reduction in the number of states that have to be explored by the machine and its client analyses.
- As clients of the extended P(CEK*)S machine, we formulate a race condition and a deadlock detection analysis. We compare two variants of each analysis: one for `cas` and one for locks. We demonstrate that the latter are more straightforward to formulate.

2. Input Language

The language we analyze is a Scheme-like language with concurrency features. Figure 1 depicts the core of this language, which we will call *Concurrent Scheme* or CScheme. It consists of atomic and complex expressions. The former can be evaluated in a bounded number of steps and do not cause side-effects, while the latter may require an unbounded number of steps to evaluate and may perform side-effects [16]. Similar to Scheme, functions can take multiple arguments ($v_1 \ldots v_n$), and `letrec` can introduce multiple bindings at once.

$$
\begin{aligned}
v &\in \mathit{Var} && \text{a set of identifiers}\\
n &\in \mathit{Num} && \text{a set of number literals}\\
b &\in \mathit{Bool} ::= \texttt{\#t} \mid \texttt{\#f}\\
e &\in \mathit{Exp} ::= æ \mid \mathit{cexp}\\
f, æ &\in \mathit{AExp} ::= \mathit{lam} \mid v \mid n \mid b\\
\mathit{lam} &\in \mathit{Lam} ::= \texttt{(lambda (}v_1 \ldots v_n\texttt{)}\ e\texttt{)}\\
\mathit{cexp} &\in \mathit{CExp} ::= \texttt{(}f\ e_1 \ldots e_n\texttt{)}\\
&\qquad \mid \texttt{(begin}\ e_1 \ldots e_n\texttt{)}\\
&\qquad \mid \texttt{(letrec ((}v_1\ e_1\texttt{)} \ldots \texttt{(}v_n\ e_n\texttt{))}\ e_{body}\texttt{)}\\
&\qquad \mid \texttt{(if}\ e_{cond}\ e_{cons}\ e_{alt}\texttt{)}\\
&\qquad \mid \texttt{(set!}\ v\ e\texttt{)}\\
&\qquad \mid \texttt{(spawn}\ e\texttt{)}\\
&\qquad \mid \texttt{(join}\ æ\texttt{)}
\end{aligned}
$$

Figure 1: Grammar of CScheme.

The `spawn` operator takes an expression that is to be evaluated by a new thread of which it returns the thread identifier. The `join` operator expects such a thread identifier as its argument, waits for the corresponding thread to complete, and returns the value computed by the thread. For clarity reasons, and without loss of generality, the argument to `join` is an atomic expression. This precludes the need to account for thread interleavings in its semantics.

The CScheme language lacks support for specifying rendezvous points between threads. Following the original description of the P(CEK*)S machine in Might and Van Horn [27], the language is extended with an *atomic compare-and-swap* (`cas`) primitive as depicted in Figure 2, resulting in the CScheme$_C$ language.

$$
\begin{aligned}
\mathit{cexp} &\in \mathit{CExp} ::= \ldots\\
&\qquad \mid \texttt{(cas}\ v\ æ_{old}\ æ_{new}\texttt{)}
\end{aligned}
$$

Figure 2: CScheme$_C$ as a CScheme extension.

The `cas` primitive takes a variable and two atomic expressions as its arguments. If the value of v is equal to the value of $æ_{old}$, the variable is updated to take the value of $æ_{new}$ and the boolean `#t` is returned. If the value of v is not equal to the value of $æ_{old}$, v remains unmodified and the boolean `#f` is returned. All of this happens *atomically*, meaning that no other thread can step during the comparison and update.

3. An Extended P(CEK*)S Machine

We recall the original formalization of the P(CEK*)S abstract machine. We then extend this abstract machine with abstract garbage collection before adding first-class support for locks as a synchronization mechanism.

3.1 The Original P(CEK*)S Machine

The P(CEK*)S abstract machine is a CESK machine that models shared-memory threading. We restate its original formalization in Might and Van Horn [27] with minor changes. It is presented as an embedding of sequential semantics inside concurrent semantics. This distinction allows us to vary the concurrent semantics without touching the underlying sequential behavior, or vice versa, for example when extending the sequential machine with additional synchronization primitives as in Section 3.3.

3.1.1 Sequential semantics

States explored by the sequential CESK machine consist of the traditional control, environment, store, and continuation components. Following the AAM approach [33], continuations are allocated in the store. Hence, they are represented as addresses inside states. The sequential CESK defines the semantics for the sequential constructs of the input language, including atomic `cas`. Figure 3 depicts the state space for the sequential CESK. The transition rule for the sequential language constructs ($(\widehat{\rightarrow}) \subset \Sigma_{\text{CESK}} \times \Sigma_{\text{CESK}}$) and the injection function ($\hat{\mathcal{I}}_{\text{CESK}} : \mathit{Exp} \rightarrow \hat{\Sigma}_{\text{CESK}}$) are detailed in the AAM literature [33].

$$
\begin{aligned}
\hat{\varsigma}_{\text{CESK}} \in \hat{\Sigma}_{\text{CESK}} &= \widehat{\mathit{Control}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Addr}}\\
\widehat{\mathit{Control}} &= \mathit{Exp} + \widehat{\mathit{Val}}\\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathit{Var} \rightharpoonup \widehat{\mathit{Addr}}\\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \rightharpoonup \mathcal{P}(\widehat{\mathit{Val}})\\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathit{Lam} \times \widehat{\mathit{Env}}\\
\widehat{\mathit{val}} \in \widehat{\mathit{Val}} &= \widehat{\mathit{Clo}} + \mathit{Bool} + \widehat{\mathit{Num}} + \widehat{\mathit{Kont}} + \widehat{\mathit{Addr}}\\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \ \ldots \mid \textbf{halt}\\
\hat{a} \in \widehat{\mathit{Addr}} & \qquad \text{a finite set of addresses}
\end{aligned}
$$

Figure 3: State space explored by the sequential CESK machine.

Atomic Evaluation Function Atomic expressions are evaluated by the atomic evaluation function $\widehat{\mathcal{E}} : \mathit{AExp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \rightarrow \mathcal{P}(\widehat{\mathit{Val}})$:

$$
\begin{aligned}
\widehat{\mathcal{E}}(n, \hat{\rho}, \hat{\sigma}) &= \{n\}\\
\widehat{\mathcal{E}}(b, \hat{\rho}, \hat{\sigma}) &= \{b\}\\
\widehat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v))\\
\widehat{\mathcal{E}}(\mathit{lam}, \hat{\rho}, \hat{\sigma}) &= \{(\mathit{lam}, \hat{\rho})\}
\end{aligned}
$$

Transition for ***`cas`*** The support of synchronization primitive `cas` is added at the level of the CESK transition relation ($\widehat{\rightarrow}$), since the semantics of `cas` only depend on the thread evaluating it. The function $\gamma : \widehat{\mathit{Val}} \rightarrow \mathcal{P}(\mathit{Val})$ is the concretization function for the abstract values, and gives the set of concrete values corresponding to one abstract value. Using the concretization function, we define the rule for `cas` as follows:

$$
\begin{aligned}
&\langle \texttt{(cas}\ v\ æ_{old}\ æ_{new}\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{a}\rangle\\
&\quad \widehat{\rightarrow} \langle \{\texttt{\#t}\}, \hat{\rho}, \hat{\sigma} \sqcup [\hat{\rho}(v) \mapsto \widehat{\mathcal{E}}(æ_{new}, \hat{\rho}, \hat{\sigma})], \hat{a}\rangle\\
&\qquad \text{unless } \hat{\sigma}(\hat{\rho}(v)) \sqcap \widehat{\mathcal{E}}(æ_{old}, \hat{\rho}, \hat{\sigma}) = \bot\\
&\quad \widehat{\rightarrow} \langle \{\texttt{\#f}\}, \hat{\rho}, \hat{\sigma}, \hat{a}\rangle\\
&\qquad \text{unless } (\hat{\sigma}(\hat{\rho}(v)) = \widehat{\mathcal{E}}(æ_{old}, \hat{\rho}, \hat{\sigma}) = \{\widehat{\mathit{val}}\}\\
&\qquad\qquad \text{and } \gamma(\widehat{\mathit{val}}) = \{\mathit{val}\})
\end{aligned}
$$

This rule leads to two possible successor states: a success state in which the swap of the value associated with the variable's address has been performed, and a failure state in which it has not. We are sure that two abstract values are equal if they correspond to the same concrete value, in which case only the success state is generated. If there is no intersection between the possible sets of values of the old and new abstract value, we know that the values are different and the machine transitions to the failure state. In all other cases, the machine transitions to both the success and the failure state.

3.1.2 Concurrent semantics

The P(CEK*)S machine models the concurrent features of our semantics, and falls back to an underlying sequential machine for single-threaded sequential semantics. It keeps a mapping of thread ids to threads, and defines the semantics of the concurrency primitives in the input languages (`spawn` and `join`). The context required for thread execution is the same context as for the sequential semantics, except for the store, which now becomes shared between all threads. Figure 4 depicts the state space for the complete multi-threaded language, explored by the P(CEK*)S machine. Note that elements of $\widehat{TID}$ should be distinguishable from $\widehat{Addr}$.

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \widehat{Threads} \times \widehat{Store} \\
\hat{T} \in \widehat{Threads} &= \widehat{TID} \rightharpoonup \mathcal{P}(\widehat{Context}) \\
\hat{c} \in \widehat{Context} &= \widehat{Control} \times \widehat{Env} \times \widehat{Addr} \\
\widehat{tid} \in \widehat{TID} & \quad \text{a finite set of thread ids, included in } \widehat{Addr}
\end{aligned}$$

Figure 4: State space explored by the P(CEK*)S machine. The unspecified components remain the same as for the CESK machine.

The P(CEK*)S machine moves between sequential and concurrent semantics: the sequential semantics are modelled by the underlying CESK machine, while the transition function of the P(CEK*)S machine adds support for thread-related operations. For this purpose we define two conversion functions. Function $\widehat{\mathcal{S}} : \widehat{Context} \times \widehat{Store} \to \hat{\Sigma}_{\text{CESK}}$ transforms a P(CEK*)S context plus shared store into an individual CESK state, and function $\widehat{\mathcal{C}} : \hat{\Sigma}_{\text{CESK}} \to \widehat{Context} \times \widehat{Store}$ does the opposite.

$$\begin{aligned}
\widehat{\mathcal{S}}(\langle e, \hat{\rho}, \hat{a}\rangle, \hat{\sigma}) &= \langle e, \hat{\rho}, \hat{\sigma}, \hat{a}\rangle \\
\widehat{\mathcal{C}}(\langle e, \hat{\rho}, \hat{\sigma}, \hat{a}\rangle) &= (\langle e, \hat{\rho}, \hat{a}\rangle, \hat{\sigma})
\end{aligned}$$

Injection Function The injection function $\widehat{\mathcal{I}} : Exp \to \hat{\Sigma}$ makes use of the injection function of the underlying CESK machine, $\widehat{\mathcal{I}}_{\text{CESK}} : Exp \to \hat{\Sigma}_{\text{CESK}}$, to inject an expression into an initial P(CEK*)S state.

$$\begin{aligned}
\widehat{\mathcal{I}}(e) &= \langle [\widehat{tid}_0 \mapsto \{\hat{c}\}], \hat{\sigma}\rangle \\
\text{where } (\hat{c}, \hat{\sigma}) &= \widehat{\mathcal{C}}(\widehat{\mathcal{I}}_{\text{CESK}}(e))
\end{aligned}$$

P(CEK*)S Transition Relation The transition relation of the P(CEK*)S machine, $(\widehat{\Rrightarrow}) \subset \hat{\Sigma} \times \widehat{TID} \times \hat{\Sigma}$, is defined in terms of the transition function of the CESK machine, $(\widehat{\rightarrow}) \subset \hat{\Sigma}_{\text{CESK}} \times \hat{\Sigma}_{\text{CESK}}$. We write $\hat{\varsigma} \overset{\widehat{tid}}{\Rrightarrow} \hat{\varsigma}'$ to denote that $(\hat{\varsigma}, \widehat{tid}, \hat{\varsigma}') \in (\widehat{\Rrightarrow})$. There are four rules that define the concurrent transition relation.

1. If one of the threads of the machine can perform a step (according to the CESK transition relation), then the P(CEK*)S machine can also perform the corresponding step. This rule is non-deterministic: if more than one thread can step, it can be used to step any of the threads, and is used for every steppable thread during the analysis.

$$\begin{aligned}
&\langle \hat{T}[\widehat{tid} \mapsto \{\hat{c}\} \cup \hat{C}], \hat{\sigma}\rangle \overset{\widehat{tid}}{\Rrightarrow} \langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\hat{c}'\}], \hat{\sigma}'\rangle \\
&\quad \text{if } \widehat{\mathcal{S}}(\hat{c}, \hat{\sigma}) \mathrel{\widehat{\rightarrow}} \hat{\varsigma}_{\text{CESK}} \text{ and } (\hat{c}', \hat{\sigma}') = \widehat{\mathcal{C}}(\hat{\varsigma}_{\text{CESK}})
\end{aligned}$$

2. When a thread halts, its final value is saved in the store, at the address corresponding to the thread identifier.

$$\begin{aligned}
&\langle \hat{T}', \hat{\sigma}\rangle \overset{\widehat{tid}}{\Rrightarrow} \langle \hat{T}', \hat{\sigma} \sqcup [\widehat{tid} \mapsto \{\widehat{val}\}]\rangle \\
&\text{where } \hat{T}' = \hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}_{\textbf{halt}}\rangle\}]
\end{aligned}$$

3. To evaluate `spawn`, the machine creates a new thread with the given expression as control component.

$$\begin{aligned}
&\langle \hat{T}[\widehat{tid}_1 \mapsto \{\overbrace{\langle (\texttt{spawn } e), \hat{\rho}, \hat{a}\rangle}^{\hat{c}}\} \cup \hat{C}], \hat{\sigma}\rangle \\
&\quad \overset{\widehat{tid}_1}{\Rrightarrow} \langle \hat{T} \sqcup [\widehat{tid}_1 \mapsto \{\hat{c}_1\}, \widehat{tid}_2 \mapsto \{\hat{c}_2\}], \hat{\sigma}\rangle \\
&\text{where } \widehat{tid}_2 = \widehat{newtid}(\hat{c}, T[\widehat{tid}_1 \mapsto \{\hat{c}\} \cup \hat{C}]) \\
&\qquad \hat{c}_1 = \langle \widehat{tid}_2, \hat{\rho}, \hat{a}\rangle \\
&\qquad \hat{c}_2 = \langle e, \hat{\rho}, \hat{a}_{\textbf{halt}}\rangle
\end{aligned}$$

4. A `join` can only be evaluated when the thread we join on has finished its execution. If this is the case, the value computed by the thread is returned.

$$\begin{aligned}
&\langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\overbrace{\langle (\texttt{join } æ), \hat{\rho}, \hat{a}\rangle}^{\hat{c}}\}], \hat{\sigma}\rangle \\
&\qquad \overset{\widehat{tid}}{\Rrightarrow} \langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}\rangle, \hat{c}\}], \hat{\sigma}\rangle \\
&\quad \text{if } \hat{\sigma}(\hat{a}_v) = \widehat{val} \\
&\quad \text{where } \hat{a}_v \in \widehat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma})
\end{aligned}$$

In order to have a monotonically increasing mapping of thread ids to threads, for the analysis to be finite, the concurrent transition rules weakly update the thread map through a join operation. If it can be determined that an abstract thread id corresponds to exactly one concrete thread, thread ids can be strongly updated, leading to increased precision [27]. Additionally, when a thread finishes its execution (rule 2), it can be safely removed from the thread map. The abstract interpreters we used in our experiments (Section 5) implement these optimizations.

Evaluation Function The evaluation function $\widehat{eval} : Exp \to \mathcal{P}(\hat{\Sigma})$ uses the transition relation to explore the state space. It computes the set of states that are reachable from an expression, using the transitive closure of $(\widehat{\Rrightarrow})$.

$$\begin{aligned}
&\widehat{eval}(e) = \{\hat{\varsigma} \mid \widehat{\mathcal{I}}(e) \widehat{\Rrightarrow}^* \hat{\varsigma}\} \\
&\text{where: } \hat{\varsigma} \widehat{\Rrightarrow}^* \hat{\varsigma}
\end{aligned}$$

$$\hat{\varsigma} \widehat{\Rrightarrow}^* \hat{\varsigma}'' \text{ iff } \hat{\varsigma} \overset{\widehat{tid}}{\Rrightarrow} \hat{\varsigma}' \wedge \hat{\varsigma}' \widehat{\Rrightarrow}^* \hat{\varsigma}''$$

It is often useful to reason about the state *graph* instead of the set of reachable states. The state graph is represented by a set of vertices, connected by edges that are annotated with a thread id, that is, $Vertices = \mathcal{P}(\hat{\Sigma})$, $Edges = \mathcal{P}(\hat{\Sigma} \times \widehat{TID} \times \hat{\Sigma})$, and $Graph = Vertices \times Edges$. This graph is computed by the function $\widehat{geval} : Exp \to Graph$. This function uses the helper function $G : \mathcal{P}(\hat{\Sigma}) \times \mathcal{P}(\hat{\Sigma}) \times Graph \to Graph$ which performs an exploration of the state space generated by the P(CEK*)S transition function in order to build the state graph.

$$\begin{aligned}
\widehat{geval}(e) &= G(\{\widehat{\mathcal{I}}(e)\}, (\{\}, \{\})) \\
G(\{\}, S, (V, E)) &= (V, E) \\
G(\{\hat{\varsigma}\} \uplus T, S, (V, E)) &= G(T, S, (V, E)) \text{ if } \hat{\varsigma} \in S \\
\text{otherwise} \quad & G(T, \{\hat{\varsigma}\} \cup S, \\
& (V \cup \{\hat{\varsigma}' \mid \hat{\varsigma} \overset{\widehat{tid}}{\Rightarrow} \hat{\varsigma}'\}, \\
& E \cup \{(\hat{\varsigma}, \widehat{tid}, \hat{\varsigma}') \mid \hat{\varsigma} \overset{\widehat{tid}}{\Rightarrow} \hat{\varsigma}'\}))
\end{aligned}$$

In Section 4 we express how properties such as the presence of a race-condition and deadlock in a program can be inferred by analyzing this state space and state graph.

3.2 Incorporating Garbage Collection

Because the P(CEK*)S machine has a non-deterministic transition rule that branches at each step to take into account every possible thread interleaving, it will generate a large number of states, although the state space remains finite. It is therefore important to find techniques that reduce the state space explored by this machine. Abstract garbage collection [26] is a state space reduction technique that aims to increase the performance of abstract interpreters. It has not yet been applied in the context of the P(CEK*)S machine, which is what we explore in this section.

Abstract garbage collection is a two-phase process: first the set of reachable addresses is determined, then the store is narrowed to only those addresses, thereby erasing all non-reachable addresses. In a sequential CESK machine, the computation of the reachable addresses is performed in the context of a single state with all its components. In the P(CEK*)S machine, however, we have as many CESK states as there are threads, which can and usually will be more than one. Because the CESK states share the same store, garbage collection has to be adapted such that it will only reclaim addresses that are unreachable by *all* threads.

If we have a function $\mathcal{R} : \Sigma_{\text{CESK}} \to \mathcal{P}(Addr)$ that computes the set of reachable addresses for a CESK state (such as the one in Might and Shivers [26]), we can define the transition relation $(\Rightarrow') : \Sigma \times \Sigma$ that performs abstract garbage collection on a P(CEK*)S state by computing the set of reachable addresses as the union of the set of addresses reachable by each thread.

$$\langle \hat{T}, \hat{\sigma} \rangle \Rightarrow' \langle \hat{T}, \hat{\sigma} | \mathcal{L} \rangle$$
$$\text{where } \mathcal{L} = \bigcup_{\hat{c} \in range(\hat{T})} \mathcal{R}(\mathcal{S}(\hat{c}, \hat{\sigma}))$$

This transition relation can be used at any point in the execution of the P(CEK*)S machine to reclaim unreachable addresses. This will lead to more free addresses, less allocation to the same addresses, and therefore may lead to an improved precision as well as a reduced state space. The reason is that some unreachable states will not be generated anymore.

However, as we will observe in Section 5.1, the impact of abstract garbage collection on the state space is not as pronounced as it is on a sequential machine [26]. We therefore investigate another possible source of state space reduction: changing the synchronization primitive for which the abstract interpreter has first-class support.

3.3 Adding First-Class Support for Locks

In the original description of the P(CEK*)S machine, `cas` is the only synchronization primitive supported by the machine. This choice enables supporting a wide variety of programs, as other synchronization primitives can be built on top of `cas`. For example, a simple implementation of locks is the following.

```
(letrec ((a-lock #f)
         (acquire (lambda ()
                    (if (cas a-lock #f #t)
                        nil
                        (acquire))))
         (release (lambda ()
                    (set! a-lock #f))))
  ...)
```

A lock is considered locked when its value is `#t`. The `acquire` function will actively try to change the value of the lock from `#f` to `#t` until it eventually succeeds, meaning that the lock was free and that it has now been updated to be locked. The `release` function will simply release the lock by setting it to the unlocked value.

As many concurrent programs make use of locks, it is important to support them. However, we claim that supporting locks as a library built on top of `cas` can induce a negative impact on the size of the state space generated by the P(CEK*)S machine, and that adding first-class support for locks (i.e., locks implemented directly in the abstract interpreter instead of build on top of `cas`) will reduce the size of the generated state space. This is because with first-class support for locks, we can formally encode the blocking nature of locks, which is not possible with `cas`. A failing `cas` that is retried until it succeeds generates many new states, whereas a blocking `acquire` prevents a thread from generating new states until the lock is available.

Extending the Language Figure 5 depicts how we extend the base language to support locks, resulting in the CScheme$_L$ language.

$$\begin{aligned}
l \in Lock &::= \texttt{\#locked} \mid \texttt{\#unlocked} \\
f, æ \in AExp &::= \ldots \mid l \\
cexp \in CExp &::= \ldots \\
&\mid \texttt{(acquire } v\texttt{)} \\
&\mid \texttt{(release } v\texttt{)}
\end{aligned}$$

Figure 5: CScheme$_L$ as a CScheme extension.

The `acquire` operation is a blocking construct that blocks the current thread until the lock given as argument becomes available, and then acquires this lock and returns. The `release` operation releases an acquired lock, making it available to blocked and future `acquire` operations.

Extending the CESK machine The support for locks is added at the level of the CESK machine, because even though locks are used in a multi-threaded setting, their semantics does not depend on the fact that there can be multiple threads acting upon them. The CESK state space is extended to support locks as values (Figure 6).

$$\widehat{val} \in \widehat{Val} = \cdots + \widehat{Lock}$$

Figure 6: CESK state space extension to support locks

CESK Transition for Locks In order to add first-class support for locks in the P(CEK*)S machine, we extend the CESK transition function to support `acquire` and `release`.

- When evaluating an `acquire`, the machine can only step if the corresponding lock is available, and then acquires the lock. Otherwise, the machine is stuck. From the perspective of a

P(CEK*)S machine, this would represent a thread waiting on a lock, and the thread will be live again when another thread releases the lock.

$$\begin{aligned}&\langle (\texttt{acquire}\ v), \hat{\rho}, \hat{\sigma}, \hat{a}\rangle \\ &\quad \widehat{\rightarrow}\ \langle \texttt{\#t}, \hat{\rho}, \hat{\sigma}[\hat{\rho}(v) \mapsto \{\texttt{\#locked}\}], \hat{a}\rangle \\ &\quad\ \text{if } \hat{\sigma}(\hat{\rho}(v)) \sqcap \{\texttt{\#unlocked}\} \neq \bot\end{aligned}$$

- The rule for `release` releases the lock by setting it to the unlocked value.

$$\begin{aligned}&\langle (\texttt{release}\ v), \hat{\rho}, \hat{\sigma}, \hat{a}\rangle \\ &\quad \widehat{\rightarrow}\ \langle \texttt{\#t}, \hat{\rho}, \hat{\sigma}[\hat{\rho}(v) \mapsto \{\texttt{\#unlocked}\}], \hat{a}\rangle\end{aligned}$$

With these additions, programs making use of locks are directly supported by the P(CEK*)S machine without needing to encode the locks with `cas`. The main advantage of this extension is that `acquire` is now a blocking operation. When `acquire` is encoded in terms of `cas`, a thread trying to acquire a lock will still generate new states as the `cas` fails and is tried again. With first-class support for locks however, the thread is *stuck* until the lock becomes available. It cannot generate new states. When the lock becomes available, the waiting thread can continue its progress.

3.4 Soundness

Theorem 1. *The abstract P(CEK*)S machine performs a sound simulation of its concrete semantics.*

Proof. Soundness has been proven for the `cas` variant of the P(CEK*)S machine [27]. Adapting this proof to the lock variant requires two adjustments. First, we need to define the concrete transition rules for evaluating `acquire` and `release`. They are trivially adapted from the abstract rules of Section 3.3 by taking into account the fact that addresses are infinite: the store therefore becomes a mapping from addresses to concrete values (instead of *sets* of abstract values), and the condition of the rule for `acquire` becomes a simple equality check. Then, since the proof is a case analysis, the cases where a context evaluates (`acquire` v) and (`release` v) have to be added. The theorem holds trivially from the definition of the transition rule. □

4. Client Analyses

We now introduce two static analyses that are built on top of the P(CEK*)S machine: one for detecting race conditions (Section 4.2), and one for detecting deadlocks (Section 4.3). The race condition analysis is based on a conflict analysis inspired by the original may-happen-in-parallel analysis [27], which we extend to avoid a specific case of false positives in Section 4.1. We compare the `cas`-based and lock-based variants for both bug detectors, and demonstrate that the lock-based detectors are more straightforward to formulate.

While the abstract interpretation is sound, this might not be the case for client analyses derived from the output of the abstract interpreter. A client analysis $C_P \subseteq \mathit{Exp}$ deciding whether a property $P \subseteq \mathit{Exp}$ (e.g., "contains a race condition") *may* hold for a program can be:

- *sound* if it contains no false negatives, i.e., $\forall e \in \mathit{Exp}, C_P(e) \Rightarrow P(e)$, or
- *complete* if it contains no false positives, i.e., $\forall e \in \mathit{Exp}, P(e) \Rightarrow C_P(e)$.

However, any finite static analysis cannot be both sound and complete. Static analyses therefore sacrifice completeness, while trying to keep an acceptable precision. That is, the number of programs e for which $P(e) \Rightarrow C_P(e)$ does not hold should be minimized in order to maintain a high precision. Precision and soundness tend not to mix well: an analysis can often gain precision by sacrificing soundness, but then becomes subject to false positives, which are problematic in the context of critical systems. Like most practical static tools [3], ours implement a particular trade-off between soundness and precision that leans toward the latter. This is because end users carry the burden of investigating every reported error.

4.1 May-Happen-in-Parallel Analysis

A may-happen-in-parallel (MHP) analysis accompanies the original abstract P(CEK*)S machine. Expressions e_1 and e_2 may happen in parallel in program e if at some point during execution of e there is one thread that evaluates e_1 while another thread evaluates e_2. This analysis is the foundation of our conflict analysis, and we formulate it as follows.

$$\begin{gathered} \mathit{MHP}_e(e_1, e_2) \Leftrightarrow \\ \exists \langle \hat{T}, \hat{\sigma}\rangle \in \widehat{eval}(e)\ \exists \widehat{tid}_1, \widehat{tid}_2 \in dom(\hat{T}), \\ \overbrace{\langle e_1, \_, \_\rangle}^{\hat{c}_1} \in \hat{T}(\widehat{tid}_1) \wedge \overbrace{\langle e_2, \_, \_\rangle}^{\hat{c}_2} \in \hat{T}(\widehat{tid}_2) \\ \wedge\ \hat{c}_1 \neq \hat{c}_2 \end{gathered}$$

Our formulation of *MHP* differs from the original in Might and Van Horn [27] in that we extended the definition with the additional conjunct requiring that $\hat{c}_1 \neq \hat{c}_2$. Without this last conjuct, if there exists a context $\hat{c}_1$ associated with $\widehat{tid}_1$ that evaluates expression e, then $\mathit{MHP}(e, e)$ would always report e to possibly happen in parallel with itself (take $\widehat{tid}_1 = \widehat{tid}_2$ and $\hat{c}_1 = \hat{c}_2$). Because two different contexts for evaluating the same expression e may be associated with the same abstract thread identifier, requiring $\widehat{tid}_1 \neq \widehat{tid}_2$ would be too restrictive, but $\hat{c}_1 \neq \hat{c}_2$ is appropriate.

This extension renders the MHP analysis more precise, but also unsound, because we can never know for sure that for the case where $\hat{c}_1 = \hat{c}_2$, this context corresponds to a single concrete thread. This analysis can therefore lead to false negatives, identifying two expressions as *will not happen in parallel*, while they may in fact happen in parallel. However, this should only happen in a very limited number of cases, when the whole execution context of each concrete thread is the same (since the continuation part of the context should match). We have not been able to find an example case where this analysis would lead to false negative. This is a case where we voluntarily lose soundness in order to gain precision: this precision gain allows us to formulate our race condition analysis (where we need to know if two write expressions, possibly equivalent ones, may happen in parallel), and only introduces a very specific case of false negatives, that does not occur in most programs.

4.2 Race Condition Detection

A race condition occurs when the value of a shared variable depends on the order in which threads are interleaved during program execution.

4.2.1 Detecting Race Conditions: `cas` Variant

There are two reasons for race conditions to happen in programs that use `cas`.

1. A race condition can be the result of a read-write conflict or a write-write conflict. If at some point the execution can continue either by first performing a read followed by a write on the same address, or by performing the write before the read, then there is a read-write conflict. A write-write conflict is the result of two concurrent writes that can happen on the same address.

2. If a cas returns false, it means that the update failed and should be retried. If a failing cas is never retried, this may lead to a race condition if the cas could have succeeded with a different thread interleaving.

Extracting Reads and Writes Because we need to reason over reads and writes that occur during program evaluation, we define two relations $\mathit{Read} \subset \widehat{\mathit{Context}} \times \widehat{\mathit{Addr}}$ and $\mathit{Write} \subset \widehat{\mathit{Context}} \times \widehat{\mathit{Addr}}$. They respectively describe the language constructs that perform a read or write operation, as well as the address touched by this operation.

$$\begin{aligned}&\mathit{Read}(\langle v, \hat{\rho}, \_\rangle, \hat{\rho}(v))\\ &\mathit{Write}(\langle (\texttt{set!}\ v\ \_), \hat{\rho}, \_\rangle, \hat{\rho}(v))\\ &\mathit{Write}(\langle (\texttt{cas}\ v\ \_\ \_), \hat{\rho}, \_\rangle, \hat{\rho}(v))\end{aligned}$$

Detecting Conflicts An execution contains a read-write conflict if at some point the execution can continue either by first performing, on the same address, a read followed by a write or a write followed by a read. Similarly, an execution contains a write-write conflict when the execution is faced with two possible orderings for writing to the same address.

The relations $\mathit{RWConflict} \subset \mathit{Exp}$ and $\mathit{WWConflict}(e) \subset \mathit{Exp}$ are inspired by the MHP relation and define that an expression contains a read-write conflict (resp. write-write conflict) if a read and a write (resp. two writes) on the same address may happen in parallel. We take special care to avoid detecting writes on the same address from the same thread in the same way as for MHP, by requiring $\hat{c}_1 \neq \hat{c}_2$. This condition is not needed for read-write conflicts, since a read expression and a write expression can never be equal to each other.

$$\begin{aligned}\mathit{RWConflict}(e) \Leftrightarrow\ & \exists\langle\hat{T}, \hat{\sigma}\rangle \in \widehat{\mathit{eval}}(e), \exists\widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}),\\ & \hat{c}_1 \in \hat{T}(\widehat{\mathit{tid}}_1) \wedge \mathit{Read}(\hat{c}_1, \hat{a}) \wedge\\ & \hat{c}_2 \in \hat{T}(\widehat{\mathit{tid}}_2) \wedge \mathit{Write}(\hat{c}_2, \hat{a})\\ \mathit{WWConflict}(e) \Leftrightarrow\ & \exists\langle\hat{T}, \hat{\sigma}\rangle \in \widehat{\mathit{eval}}(e), \exists\widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}),\\ & \hat{c}_1 \in \hat{T}(\widehat{\mathit{tid}}_1) \wedge \mathit{Write}(\hat{c}_1, \hat{a}) \wedge\\ & \hat{c}_2 \in \hat{T}(\widehat{\mathit{tid}}_2) \wedge \mathit{Write}(\hat{c}_2, \hat{a}) \wedge\\ & \hat{c}_1 \neq \hat{c}_2\end{aligned}$$

These two relations are combined in the $\mathit{Conflict} \subset \mathit{Exp}$ relation.

$$\mathit{Conflict}(e) \Leftrightarrow\ \mathit{RWConflict}(e) \vee \mathit{WWConflict}(e)$$

Filtering Harmless Conflicts The result of the conflict analysis will contain harmless conflicts that involve cas. This is because cas is often used in a loop that retries cas until it succeeds, as illustrated by the following program.

```
(letrec ((counter 0)
         (inc (lambda ()
                (letrec ((old counter)
                         (new (+ old 1)))
                  (if (cas counter old new)
                    #t
                    (inc)))))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  ;; will always be 2 (2 calls to inc)
  counter)
```

With multiple concurrent calls to inc, the analysis would detect a write-write conflict on cas in the example program. But this conflict is harmless, because in this particular case one cas will succeed, and the other will be retried and eventually also succeed. Similarly, the conflict analysis would also detect a harmless read-write conflict due to binding old and evaluating cas in parallel. To avoid detection of harmless conflicts, our detector extracts the addresses and expressions involved in the conflict, and filters out some conflicts. Filtering happens *after* having found conflicts. For each address $\hat{a}$ involved in such a conflict, and for each pair of threads involved in the conflicts on $\hat{a}$, we can ignore conflicts involving $\hat{a}$ and the two threads if the only conflicts detected are both of the following, or only the latter, and no other conflict is detected for this address.

1. A read-write conflict between the evaluation of a variable living at address $\hat{a}$ and a cas on the same address $\hat{a}$.
2. A write-write conflict between two cas on the same address $\hat{a}$.

Note that this filtering will only filter false positives for programs involving variables that are shared by two threads. The pattern could be extended to more threads, but we leave this extension and its implementation as future work.

Detecting Unretried **cas** In order to detect a cas that is not retried, it is no longer sufficient to look at individual P(CEK*)S states. We need to reason about relations between multiple states of the program execution, i.e., we need to explore the state graph generated by the P(CEK*)S machine through $\widehat{\mathit{geval}}$. The following relations will be useful when reasoning about this state graph.

- $\mathit{Successor} \subset \mathit{Edges} \times \hat{\Sigma} \times \hat{\Sigma}$ is the relation of states that directly follow another state.

$$\mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) \Leftrightarrow (\hat{\varsigma}_1, \_, \hat{\varsigma}_2) \in E$$

- $\mathit{Path} \subset \mathit{Edges} \times \hat{\Sigma} \times \hat{\Sigma}$ is the relation of paths that join two states.

$$\begin{aligned}\mathit{Path}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) \Leftrightarrow\ & \mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) \vee\\ & (\mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_{1'}) \wedge\\ & \ \mathit{Path}(E, \hat{\varsigma}_{1'}, \hat{\varsigma}_2))\end{aligned}$$

- $\mathit{TidExp} \subset \widehat{\mathit{Threads}} \times \widehat{\mathit{TID}} \times \mathit{Exp}$ is the relation of threads that are currently evaluating a particular expression.

$$\mathit{TidExp}(\hat{T}, \widehat{\mathit{tid}}, e) \Leftrightarrow \langle e, \_, \_\rangle \in \hat{T}(\widehat{\mathit{tid}})$$

- $\mathit{PathToTidExp} \subset \mathit{Edges} \times \hat{\Sigma} \times \widehat{\mathit{TID}} \times \mathit{Exp}$ is the relation of paths that evaluate a given expression on any thread.

$$\begin{aligned}\mathit{PathToTidExp}(E, \hat{\varsigma}, \widehat{\mathit{tid}}, e) \Leftrightarrow\ & \mathit{Path}(E, \hat{\varsigma}, \langle\hat{T}', \hat{\sigma}'\rangle) \wedge\\ & \mathit{TidExp}(\hat{T}', \widehat{\mathit{tid}}, e)\end{aligned}$$

Using these relations, we can construct our analysis for detecting failed but unretried cas operations in programs. A failed cas is not retried if there exists no path from the resulting failure state to a state evaluating that same cas again. This means that every correct use of cas has to loop to itself to ensure that the update is eventually performed in case of failure. Figure 7 depicts the pattern we want to detect in $\widehat{\mathit{geval}}$.

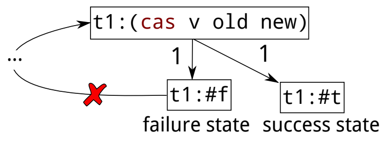

Figure 7: Pattern searched for by the unretried cas analysis for CScheme$_C$.

We can now define the relation $\mathit{UnretriedCas}(e) \subset \mathit{Exp}$ that encodes our analysis of unretried cas:

$$\begin{aligned}\mathit{UnretriedCas}(e) \Leftrightarrow\ & (V, E) = \widehat{\mathit{geval}}(e) \wedge \\ & \exists \langle \hat{T}_1, \hat{\sigma}_1 \rangle \in V, \exists \widehat{\mathit{tid}} \in \mathit{dom}(\hat{T}_1)\ \wedge \\ & \mathit{TidExp}(\hat{T}_1, \widehat{\mathit{tid}}, \overbrace{\texttt{(cas \_ \_ \_)}}^{e}) \wedge \\ & \mathit{Successor}(E, \langle \hat{T}_1, \hat{\sigma}_1 \rangle, \langle \hat{T}_2, \hat{\sigma}_2 \rangle) \wedge \\ & \mathit{TidExp}(\hat{T}_2, \widehat{\mathit{tid}}, \alpha(\texttt{\#f})) \wedge \\ & \neg \mathit{PathToTidExp}(E, \langle \hat{T}_2, \hat{\sigma}_2 \rangle, \widehat{\mathit{tid}}, e)\end{aligned}$$

***Combining** Conflict **and** UnretriedCas* Combining the conflict analysis with the unretried cas analysis results in a race condition analysis. Because any error detected by one of these analyses may lead to a race condition, the definition of $\mathit{RaceCondition} \subset \mathit{Exp}$ is straightforward:

$$\mathit{RaceCondition}(e) \Leftrightarrow \mathit{Conflict}(e) \vee \mathit{UnretriedCas}(e)$$

When this relation holds for e, there *might* exist one execution of e containing a race condition.

Since the MHP analysis is unsound, and the race detection is based on this analysis, the race detection is also unsound. Again, this unsoundness appears only in very specific and rare cases that we did not encounter during our evaluation, and allows us to have an analysis with an improved precision (see Section 5.2). Without it, the analysis results would be flooded with false positives that would render it unusable.

4.2.2 Detecting Race Conditions: Lock Variant

We now change the setting and look at programs that use locks as first-class citizens instead of cas (or locks implemented on top of cas) to synchronize between threads. We find that defining the race condition analysis for lock-based programs becomes more straightforward.

The *Write* relation now becomes the following (there is no cas in the language).

$$\mathit{Write}(\langle \texttt{(set!}\ v\ \_\texttt{)}, \hat{\rho}, \_\rangle, \hat{\rho}(v))$$

The *Conflict* relation remains the same, but is now equivalent to the *RaceCondition* relation as there is no more need for detecting unretried cas.

$$\mathit{RaceCondition}(e) \Leftrightarrow \mathit{Conflict}(e)$$

Notice that in this case, the acquire and release are not present in the definition of the race detection. This is because these constructs act on the level of the graph computed by the abstract interpreter. They prevent states to be generated when trying to acquire a held lock. The static analysis therefore only has to look for the presence of a bad state, and is independent of the fact that locks are used as the concurrency primitive, but it does depend on the fact that we are using a blocking synchronization mechanism instead of a non-blocking one like cas.

For the same reason as the cas variant, this analysis is unsound due to the unsoundness of our modified MHP analysis. Again, the unsoundness should only appear in very specific cases, and allows us to have a good precision for detecting write-write conflicts, even though some defects might not be detected. This is one case where we have to choose between a sound analysis with many false positives, rendering the analysis useless in practice, and an unsound analysis with good precision, potentially missing defects.

4.2.3 Comparison: cas Versus Locks

The race condition analysis is much more succinct and computationally less demanding for CScheme$_L$ than for CScheme$_C$. First, there is no need to check for unretried cas, which requires verifying the absence of cycles in the state graph. Also, the conflicts detected by the conflict analysis do not have to be filtered, as there is no equivalent to the harmless cas conflict if the program only uses set! for writing variables.

4.3 Deadlock Detection

A program execution reaches a deadlock when the execution no longer makes any progress. Like race condition detection before, we examine deadlock detection in the settings of cas and locks.

4.3.1 Detecting Deadlocks: cas Variant

As the cas operation can be used as a basic building block for implementing locks, it may introduce deadlocks as illustrated by the program below.

```
(letrec ((lock #t) ; lock already locked
         (f (lambda ()
              (if (cas lock #f #t)
                nil
                (f)))))
  ;; Trying to acquire the lock which is already held
  (f))
```

Listing 1: Deadlock involving cas

This example program gets stuck while trying to acquire a lock that is never released, resulting in a deadlock (technically a *livelock*). When such a deadlock occurs, the state graph will contain a loop where a thread tries to acquire a lock, fails, and goes back to trying, without ever succeeding in acquiring the lock. However, detecting this pattern will lead to many false positives for locks used by more than one thread. If two threads try to acquire the same lock and thread 1 succeeds, there exists a loop in the graph where thread 1 never gets executed again and the lock is therefore never released, preventing thread 2 from ever acquiring it. This is a possible deadlock, but any real scheduler will eventually execute thread 1 again, allowing it to release the lock and avoiding the deadlock.

If a lock is used by only one thread, deadlocks are still possible, as Listing 1 shows. For a deadlock to happen in this case, every thread – except the one trying to acquire the lock – should be stuck. The only construct possibly leading to a stuck state in CScheme$_C$ is join. Therefore, if one thread follows the loop pattern and every other thread is currently executing a join, the analysis will detect this as a deadlock. This is overly approximative, as for a deadlock to happen the joins should be dependent on the thread performing the cas, e.g., thread 3 joins on thread 2, which joins on thread 1 which is stuck in the acquire loop.

One problem remains: as the value of the lock can be an abstract value, we may not always know whether the cas succeeds or not. The resulting loop in the state graph will therefore have one exit path, which may never be reached. Unfortunately, this pattern is indistinguishable from a cas that is correctly used. We therefore refine our deadlock analysis such that it only reports deadlocks for which a loop with no exit path is found. This analysis is therefore unsound by design, in order to improve precision. Figure 8 depicts the pattern we are looking for in $\widehat{geval}$.

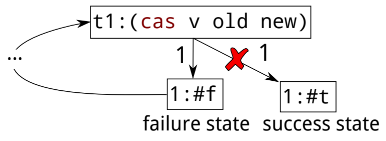

Figure 8: Pattern in the state graph of a CScheme$_C$ program indicating a deadlock.

Formalizing the deadlock analysis for CScheme$_C$ programs requires the following relations:

- $Cycle \subset Edges \times \hat{\Sigma}$ is the relation of states that cycle back to themselves in the state graph.

$$Cycle(E, \hat{\varsigma}) \Leftrightarrow Path(E, \hat{\varsigma}, \hat{\varsigma})$$

- $FailingCas \subset Edges \times \hat{\Sigma} \times \widehat{TID}$ is the relation of states that evaluate a `cas` on a certain thread, and for which the `cas` does not succeed.

$$\begin{aligned} &FailingCas(E, \langle \hat{T}, \hat{\sigma}\rangle, \widehat{tid}) \Leftrightarrow \\ &\quad TidExp(\hat{T}, \widehat{tid}, \texttt{(cas \_ \_ \_)}) \wedge \\ &\quad \neg(Successor(E, \langle \hat{T}, \hat{\sigma}\rangle, \langle \hat{T}', \hat{\sigma}'\rangle) \wedge \\ &\quad\quad TidExp(\hat{T}', \widehat{tid}, \texttt{\#t})) \end{aligned}$$

- $NumberOfNotJoins \subset \widehat{Threads} \times \widehat{TID} \times \mathbb{N}$ counts, for a given thread in a given state, the number of expressions that do *not* evaluate a `join`.

$$\begin{aligned} &NumberOfNotJoins(\hat{T}, \widehat{tid}, n) \Leftrightarrow \\ &\quad |\{\hat{c} \in \hat{T}(\widehat{tid}) \text{ s.t. } \hat{c} \neq \langle \texttt{(join \_)}, \_, \_\rangle\}| = n \end{aligned}$$

- $TransitionsOnPath \subset Edges \times \hat{\Sigma} \times \hat{\Sigma} \times \mathcal{P}(\widehat{TID})$ computes, along a path between two states, the set of threads for which a transition rule is used.

$$\begin{aligned} &TransitionsOnPath(E, \hat{\varsigma}_1, \hat{\varsigma}_2, Tr) \Leftrightarrow \\ &\quad ((\hat{\varsigma}_1, \widehat{tid}, \hat{\varsigma}_2) \in E \ \wedge \ Tr = \{\widehat{tid}\}) \vee \\ &\quad ((\hat{\varsigma}_1, \widehat{tid}, \hat{\varsigma}_{1'}) \in E \wedge \\ &\quad\quad TransitionsOnPath(E, \hat{\varsigma}_{1'}, \hat{\varsigma}_2, Tr') \wedge \\ &\quad\quad Tr = Tr' \cup \{\widehat{tid}\}) \end{aligned}$$

- $NumberOfTransitionsOnCycle \subset Edges \times \hat{\Sigma} \times \mathbb{N}$ counts, along a cycle from a state to itself, the number of different threads for which a transition rule is used.

$$\begin{aligned} &NumberOfTransitionsOnCycle(E, \hat{\varsigma}, n) \Leftrightarrow \\ &\quad TransitionsOnPath(E, \hat{\varsigma}, \hat{\varsigma}, Tr) \wedge |Tr| = n \end{aligned}$$

The deadlock analysis is formalized by the $Deadlock \subset Exp$ relation, which expresses that a deadlock is present when we have a `cas` that fails and cycles back to itself, and either every other thread is blocked, or more than one thread performs a transition along the cycle.

$$\begin{aligned} &Deadlock(e) \Leftrightarrow \\ &\quad (V, E) = \widehat{geval}(e) \wedge \exists \langle \hat{T}, \hat{\sigma}\rangle \in V, \\ &\quad \exists \widehat{tid} \in dom(\hat{T}) \wedge \\ &\quad FailingCas(E, \langle \hat{T}, \hat{\sigma}\rangle, \widehat{tid}) \wedge \\ &\quad Cycle(\langle \hat{T}, \hat{\sigma}\rangle) \wedge \\ &\quad ((\forall \widehat{tid}' \neq \widehat{tid}, \\ &\quad\quad NumberOfNotJoins(\hat{T}, \widehat{tid}', 0) \wedge \\ &\quad\quad NumberOfNotJoins(\hat{T}, \widehat{tid}, 1)) \vee \\ &\quad (\exists n > 1, \\ &\quad\quad NumberOfTransitionsOnCycle(E, \langle \hat{T}, \hat{\sigma}\rangle, n))) \end{aligned}$$

4.3.2 Detecting Deadlocks: Locks Variant

When a deadlock occurs with the CScheme$_L$ language, the state graph will contain a state which has not finished its evaluation and has no successor (since the machine is completely stuck, by definition of *deadlock*). Finding such a state is computationally less demanding than finding a deadlock for CScheme$_C$, which involved finding loops in the state graph.

A CESK state can only be stuck when evaluating `acquire` on an already held lock, or when evaluating a `join`. A P(CEK*)S state will therefore be in deadlock if the state has no successor, and if every thread is blocked either by an `acquire` or `join`.

$$\begin{aligned} Deadlock(e) \Leftrightarrow &(V, E) = \widehat{geval}(e), \exists \langle \hat{T}_1, \hat{\sigma}_1\rangle \in V \wedge \\ &\nexists(\langle \hat{T}_1, \hat{\sigma}_1\rangle, \langle \hat{T}_2, \hat{\sigma}_2\rangle) \in E \wedge \\ &\forall \widehat{cs} \in range(\hat{T}_1), \forall \hat{c} \in \widehat{cs}, \\ &(\hat{c} = \langle \texttt{(acquire \_)}, \_, \_\rangle \vee \\ &\ \hat{c} = \langle \texttt{(join \_)}, \_, \_\rangle) \end{aligned}$$

We claim that this analysis is sound. If, for some reason, in an abstract state corresponding to a concrete deadlock state, a thread id is associated with more than one abstract thread, where one thread does not evaluate either an `acquire` or `join`, then the machine can take a step. The machine will make a step on this thread (and any other similar thread, except the ones evaluating a blocking construct) until they finish their execution. When this thread finishes its execution, it is removed from the abstract state, therefore leaving only the threads evaluating blocking constructs, and the analysis will therefore detect this new state as a deadlock state. Since the *Deadlock* predicate acts upon an entire program, the program will correctly be detected as containing a deadlock.

4.3.3 Comparison: `cas` Versus Locks

The deadlock analysis for CScheme$_C$ involves looking for a cycle without exit path in the state graph, whereas the deadlock analysis for CScheme$_L$ only requires detection of stuck states. This gain in simplicity also avoids a pitfall of the analysis for CScheme$_C$: any deadlock found by the analysis for CScheme$_L$ will lead to a stuck state, whereas with CScheme$_C$ we are unable to distinguish between a deadlock and a correctly used `cas`.

5. Experimental Evaluation

Our implementation of the P(CEK*)S machine and the analyses described in the previous section is publicly available[1]. Because the P(CEK*)S machine has to explore every possible thread interleaving, the programs it can handle remain small and involve a small

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

number of threads. We nevertheless are able to evaluate the impact of our extensions to the P(CEK*)S machine. It has been shown that in practice, concurrency bugs tend to involve only a small number of threads and variables [25]. Programs selected for this evaluation are programs that use the concurrency features of the language, that are small enough to be handled by our implementation, and that exhibit various issues related to concurrent programs.

For example, the `pcounter` program models a counter shared across multiple threads, supporting an `inc` operation. There are multiple variants of this program, each corresponding to a way of using one of the synchronization primitives to build this program, some of them containing race conditions. The typical implementation of the `pcounter` program using locks is given in Listing 2.

```
(letrec ((lock #unlocked)
         (counter 0)
         (inc (lambda ()
                (acquire lock)
                (set! counter (+ counter 1))
                (release lock)))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

Listing 2: Shared counter implemented with locks.

The other benchmark programs in CScheme$_C$ include different cases of deadlocks (`deadlock1`, `deadlock1-release`, `deadlock`), programs containing race conditions due to improper use of `cas` (`race-cas`, `race-set-cas`), an implementation of the producer-consumer pattern where the producer is first executed in a thread, and when it finishes, the consumer is executed in another thread (`producer-consumer-seq`), a program containing a benign race-condition (that has no influence on the outcome of the program) between two threads (`benign`), and a program used to check whether false positives are reported due to the allocation scheme used (`false-pos`). In CScheme$_L$, the programs are similar: there are multiple variants of the shared counter, one variant with support for a decrease operation as well (`incdec`), programs containing deadlocks due to incorrect use of locks (`deadlock1`, `deadlock1-release`, `deadlock`, `deadlock3`), and due to circularity in the `joins` (`join-lock3`).

5.1 Impact of Abstract Garbage Collection

Abstract garbage collection is an important technique to reduce the number of states of sequential CESK machines, enabling support for more complex programs [26]. In Section 3.2, we showed how to incorporate it in the P(CEK*)S machine.

Tables 1 and 2 show the impact of garbage collection on the state space produced by the P(CEK*)S machine. The first group of columns represent the number of states in the state graph (n) and computation time (t) for a program with garbage collection disabled, the second group is with garbage collection enabled, and the last group is the ratio between the number of states with garbage collection enabled and disabled. We observe no consistent improvements in the precision and performance. When analyzing the CScheme$_C$ language, garbage collection sometimes yields a substantial reduction in the number of states (e.g., 97% in the case of `deadlock`), but can also increase the number of generated states (up to $+115\%$ for `pcounter`). The advantage of abstract garbage collection for CScheme$_C$ is therefore smaller than for a sequential language.

This advantage disappears for the CScheme$_L$ language. Except for `producer-consumer-seq`, which benefits of a reduction of

Table 1: Impact of GC on CScheme$_C$ programs, on the number of states (n) and the time in seconds (t). The lowest number of states and time for each benchmark are in bold.

Example	Without GC		With GC		GC/NoGC	
	n	t	n	t	n	t
`pcounter1`	305	1	**123**	ϵ	40%	–
`pcounter`	**8845**	**112**	19047	343	215%	306%
`pcounter5-seq`	1768	12	**588**	**3**	33%	25%
`pcounter-mutex`	2705	**27**	**2497**	29	92%	107%
`pcounter-race`	**457**	**3**	669	5	146%	166%
`pcounter-buggy`	**8845**	**110**	19047	293	215%	266%
`producer-consumer-seq`	796	4	**678**	4	85%	100%
`deadlock1`	1530	8	**331**	**1**	22%	13%
`deadlock1-release`	3679	23	**664**	**3**	18%	13%
`deadlock`	234962	4258	**6043**	**1368**	3%	32%
`race-cas`	81	ϵ	81	ϵ	100%	–
`race-set-cas`	95	ϵ	95	ϵ	100%	–
`false-pos`	461	4	461	4	100%	100%
`benign`	75	ϵ	75	ϵ	100%	–

Table 2: Impact of GC on CScheme$_L$ programs, on the number of states (n) and the time in seconds (t). The lowest number of states and time for each benchmark are in bold.

Example	Without GC		With GC		GC/NoGC	
	n	t	n	t	n	t
`pcounter1`	26	ϵ	26	ϵ	100%	–
`pcounter`	**443**	**2**	507	3	114%	150%
`pcounter5-seq`	200	1	200	1	100%	100%
`producer-consumer-seq`	710	3	**662**	3	93%	100%
`join-lock3`	422	4	422	4	100%	100%
`pcounter3`	**3827**	**47**	4827	64	126%	136%
`deadlock1`	12	ϵ	12	ϵ	100%	–
`deadlock1-release`	19	ϵ	19	ϵ	100%	–
`deadlock`	199	1	199	1	100%	100%
`deadlock3`	1793	**21**	1793	25	100%	119%
`incdec`	**3829**	**60**	4349	69	113%	115%

7% of its state space, every other program either maintains its number of states or suffers from an increase in number of states.

In these experiments, the expected reduction in state space resulting from abstract garbage collection are less pronounced than in the CESK setting.

5.2 Race Condition Analysis

Table 3 lists the number of true positives and false positives of the race condition analysis for CScheme$_C$, and Table 4 for CScheme$_L$. Each benchmark has a number of line of codes (*LOC*), a number of threads (*#T*), a number of expected defects (*Expected*), the number of defects found, and how many among them are true positives (*tp*) and false positives (*fp*). For CScheme$_L$, benchmarks also have a number of locks used (*#L*). Both CScheme$_C$ and CScheme$_L$ are subject to a false positive in the case of *benign* race conditions. A race condition is benign if it has no impact on the end result of a program (e.g., two concurrent updates of a shared variable to a same value). Whether this should count as a true or false positive is arguable. Except for this case, both analyses find every race condition present without any false positives. We have subjected more CScheme$_C$ programs to the `cas`-based detector, because race conditions may happen for various reasons in programs involving `cas`, whereas race conditions with locks happen only in case of conflicts. Note that the analysis for CScheme$_L$ is able to handle programs with 4 threads whereas the analysis for CScheme$_C$ is not.

5.3 Deadlock Analysis

Table 5 lists the results of the deadlock analysis for CScheme$_C$, and Table 6 for CScheme$_L$. For CScheme$_C$, two false positives are reported in one example, but due to the size of the resulting

Table 3: Race condition detector for the CScheme$_C$ language.

Example	LOC	#T	Expected	Results		
				found	tp	fp
`pcounter`	34	3	0	0	0	0
`pcounter-mutex`	45	3	0	0	0	0
`pcounter-race`	24	3	2	2	2	0
`pcounter-buggy`	34	3	3	3	3	0
`false-pos`	34	3	0	0	0	0
`benign`	17	3	0	1	0	1
`race-cas`	20	3	1	1	1	0
`race-set-cas`	19	3	1	1	1	0

Table 4: Race condition detector for the CScheme$_L$ language.

Example	LOC	#T	#L	Expected	Results		
					found	tp	fp
`pcounter`	32	3	1	0	0	0	0
`pcounter3`	38	4	1	0	0	0	0
`incdec`	52	4	1	0	0	0	0
`pcounter-race`	24	3	0	2	2	2	0
`false-pos`	34	3	0	0	0	0	0
`benign`	17	3	0	0	1	0	1

state graph, we were not able to pinpoint the reason of their detection. We conjecture that the two false positives are repetitions of the two already found deadlocks, with only small changes in the states making them different. Also, due to the fact that the analysis only looks for cycles without exit paths, one deadlock is missed in a program where the value of the lock gets abstracted. For CScheme$_L$, no deadlock is missed, no false positives are present, and the analysis is able to handle programs involving more threads and locks.

Table 5: Results of the deadlock analysis for the CScheme$_C$ language.

Example	LOC	#T	#L	Expected	Results		
					found	tp	fp
`deadlock-simple`	15	1	1	1	1	1	0
`deadlock-abstract`	18	1	1	1	0	0	0
`deadlock1`	27	3	1	2	2	2	0
`deadlock1-release`	29	2	1	1	1	1	0
`deadlock`	65	3	2	2	4	2	2
`pcounter-mutex`	45	3	1	0	0	0	0

Table 6: Results of the deadlock analysis for the CScheme$_L$ language.

Example	LOC	#T	#L	Expected	Results		
					found	tp	fp
`deadlock1`	13	2	1	2	2	2	0
`deadlock`	39	3	2	1	1	1	0
`deadlock3`	58	4	3	1	1	1	0
`deadlock1-release`	16	2	1	1	1	1	0
`pcounter`	32	3	1	0	0	0	0
`pcounter3`	38	4	1	0	0	0	0
`incdec`	52	4	1	0	0	0	0

Due to the complexity of the analysis, the P(CEK*)S machine for the CScheme$_C$ language was only able to handle examples with up to three threads (with only two concurrently accessing a shared variable) and two locks, whereas for the CScheme$_L$ language it supports programs with four threads and three locks.

5.4 State Space Comparison

Even though the different analyses can only handle small programs due to the complexity of computing every thread interleaving, our evaluation shows that the analysis performs better for CScheme$_L$ than for CScheme$_C$, in terms of complexity of the programs supported.

This is due to the fact that `cas` is a non-blocking synchronization construct, and can therefore generate many new states that are not useful for the progress of the evaluation. On the other hand, locking is a blocking mechanism that entirely stops a thread trying to acquire a held lock, therefore preventing new states from being generated by transitions on this thread. Figure 9 (raw numbers in Table 7, along with a ratio for each benchmark of the number of states generated using locks over the number of states generated using `cas`, and similarly for a ratio of the computation time) compares the number of states computed for programs implemented once with `cas` and once with locks as synchronization primitives. The reduction in number of states varies from $\div 1.12$ to $\div 1181$, and leads in most case to a reduction of more than one order of magnitude.

Table 7: State space size comparison between similar programs implemented in CScheme$_C$ and CScheme$_L$

Example	CScheme$_C$		CScheme$_L$		L/C	
	n	t	n	t	n	t
`pcounter1`	305	1	26	ϵ	8%	–
`pcounter`	8845	112	443	2	5%	2%
`pcounter5-seq`	1768	12	200	1	11%	8%
`producer-consumer-seq`	796	4	710	3	89%	75%
`deadlock1`	1530	8	12	ϵ	0.8%	<12%
`deadlock1-release`	3679	23	19	ϵ	0.5%	<4%
`deadlock`	234962	4258	199	1	0.08%	0.02%

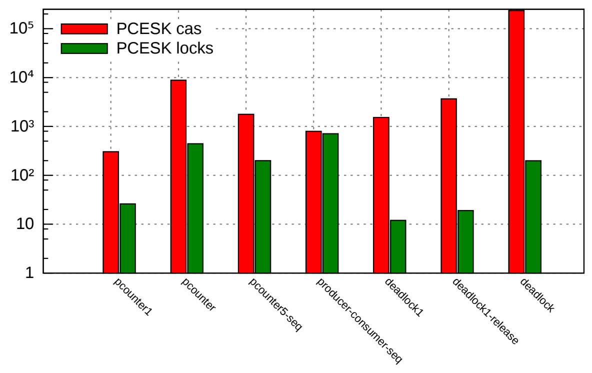

Figure 9: State size comparison between similar programs implemented in CScheme$_C$ and CScheme$_L$.

6. Related Work

Abstract Interpretation Our work is based on previous work by Might and Van Horn [27] on the P(CEK*)S machine, an adaptation of the AAM approach [33] to concurrent programs. Another approach to abstract interpretation for concurrent higher-order programs is Jagannathan et al.'s work [22, 23, 35]. Their abstract interpreter computes the possible values of each variables at each program point in order to perform compiler optimizations, but it is not adapted to reason about race conditions and deadlocks. In their abstract interpreter, synchronization is performed through blocking primitives based on *shared locations* that can be read from in a blocking manner and written to. Other synchronization mechanisms are implemented on top of these shared locations to justify the usefulness of the language, but no first-class support is added for those derived synchronization mechanisms.

Formal Verification Verification of concurrent programs is an important application of model checking techniques. Most model checking tools that support the verification of concurrent programs (e.g., SPIN [21]) require the program to be modeled into a specification language. Exceptions to this include Bandera [11], Java PathFinder [20], VeriSoft [18], and McErlang [17]. Bandera, Java PathFinder and VeriSoft are able to deduce the specification from the program source, while McErlang implements an Erlang virtual machine that supports temporal logic queries. All these tools perform verification by taking as input user-defined formulas written in temporal logic that should hold during the program execution. This is in contrast with our approach, which require no user-defined formula dependent on the verified program to detect race conditions and deadlocks. Our approach requires a formula which only depends on the semantics of the language analyzed, and not on the specific program being analyzed. On the other hand, the state space generated by the P(CEK*)S machine could be explored to verify temporal logic formulas [32].

Type Systems Multiple extension of the Java type system have been described to ensure the absence of race conditions [6, 7, 15] and deadlocks [5, 7, 14]. The resulting programs tend to require heavy type annotations to be accepted by the type checker. Further work has shown that some of the required annotations could be automatically inferred [1, 5, 31]. Approaches based on type systems will generally reject some race-free and deadlock-free programs, if those are not deducible from the types. Our approach, however, precisely approximates the control-flow of the program, making the analysis less prone to false positives.

Lock-Based Algorithms Various other techniques detect deadlocks in imperative programs by using algorithms analyzing lock usage. Such techniques are generally based on building a lock-order graph capturing the locking information of the program [2, 34, 36], or on computing the set of locks protecting each variable [13]. These techniques are only applicable to lock-based imperative programs, whereas the P(CEK*)S approach can support different synchronization mechanisms as shown in this work, and is not restricted to imperative features. Naik et al.'s approach to detect race conditions [28] and deadlocks [29] have the specificity of combining multiple static analyses to try to refute a condition for the absence of the defect. This idea of describing a defect as a disjunction of multiple smaller analyses is shared by our race condition analysis.

7. Conclusion

In this work we explored how the P(CEK*)S machine could be extended and used as a starting point for building client analyses that detect concurrency bugs. We extended this machine with first-class support for locks, as this synchronization primitive is more widely used than `cas`. Experiments show that the state space for programs containing first-class locks is generally more than one order of magnitude smaller than the state space of corresponding programs with locks implemented on top of `cas`. We also adapted and applied abstract garbage collection to the P(CEK*)S machine, but observed no consistent improvements in precision and performance. We formulated deadlock and race condition analyses based on state graphs computed by the P(CEK*)S machine. Querying state graphs was significantly simplified by using locks as synchronization mechanism instead of `cas`.

8. Future Work

We identify three ways to improve the precision and scalability of our approach.

Pushdown analyses The P(CEK*)S machine is described following the AAM approach to abstract interpretation. This approach has the disadvantage of not precisely matching calls and returns of functions. More recent techniques, such as PDCFA [12] and AAC [24], allow call-return matching with the precision offered by pushdown systems. Adapting the P(CEK*)S machine to these techniques could lead to precision improvements, but represents a substantial theoretical challenge.

State space reduction The state space computed by the P(CEK*)S machine suffers from the state space explosion problem that is also found in model checkers supporting concurrency. Research in the area of model checking has been focused on tackling this problem [10], and thanks to techniques such as partial-order reduction [19, 30], binary decision diagrams [9], and bounded model checking [4], models of up to 10^{120} states can be verified [8], and states spaces can be significantly reduced. For example [19] reduces on average 73% of the states, and up to 99% of them on some examples. Importing such techniques to abstract interpretation could yield a gain in performance and a reduction of state space.

Other concurrency primitives As moving from `cas` to locks yields an improvement in the simplicity of the analyses and the efficiency of the machine, investigating other concurrency primitives (e.g., STMs), other concurrency models (e.g., the actor model), could lead to other insights and improvements.

Acknowledgments

This work has been supported by the *GRAVE* project of the Research Foundation - Flanders (FWO). Jens Nicolay is funded by the Flemish agency for Innovation by Science and Technology (IWT Vlaanderen).

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75. IEEE, 2001.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[5] C. Boyapati. *Safejava: a unified type system for safe programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, volume 37, pages 211–230. ACM, 2002.

[8] J. Burch, E. M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. *Computer Science Department*, page 435, 1991.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. 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.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[12] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 177–188. ACM, 2012.

[13] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[14] C. Flanagan and M. Abadi. Types for safe locking. In *Programming Languages and Systems*, pages 91–108. Springer, 1999.

[15] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000.

[16] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[17] L.-Å. Fredlund and H. Svensson. Mcerlang: a model checker for a distributed functional programming language. In *ACM SIGPLAN Notices*, volume 42, pages 125–136. ACM, 2007.

[18] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[19] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. *Model Checking Software*, page 95, 2007.

[20] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[21] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[22] S. Jagannathan. Locality abstractions for parallel and distributed computing. In *Theory and Practice of Parallel Programming*, pages 320–345. Springer, 1995.

[23] S. Jagannathan and S. Weeks. Analyzing stores and references in a parallel symbolic language. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 294–305. ACM, 1994.

[24] J. I. Johnson and D. Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 11–22. ACM, 2014.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[26] M. Might and O. Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. *ACM SIGPLAN Notices*, 41(9):13–25, 2006.

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

[28] M. Naik, A. Aiken, and J. Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[29] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

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

[31] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94. ACM, 2005.

[32] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Static Analysis*, pages 351–380. Springer, 1998.

[33] D. Van Horn and M. Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[34] C. von Praun. *Detecting synchronization defects in multi-threaded object-oriented programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[35] S. Weeks, S. Jagannathan, and J. Philbin. A concurrent abstract interpreter. *Lisp and Symbolic Computation*, 7(2-3):173–193, 1994.

[36] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005-Object-Oriented Programming*, pages 602–629. Springer, 2005.