Thread-Modular Counterexample-Guided Abstraction Refinement*

Alexander Malkis[1], Andreas Podelski[2], and Andrey Rybalchenko[3]

[1] IMDEA Software
[2] University of Freiburg
[3] TU München

Abstract. We consider the refinement of a static analysis method called thread-modular verification. It was an open question whether such a refinement can be done automatically. We present a counterexample-guided abstraction refinement algorithm for thread-modular verification and demonstrate its potential, both theoretically and practically.

1 Introduction

Verification of multi-threaded programs is both important and challenging. State space explosion is a fundamental problem for any complete verification method: the state space of a program increases exponentially in the number of its threads. As a consequence, no static analysis method both always scales (i.e., is polynomial) in the number of threads and always provides a conclusive answer (i.e. never says "don't know").

Abstraction is the key approach to deal with the state space explosion. Multi-threaded Cartesian abstraction, also known as thread-modular reasoning [22] and as the Owicki-Gries method without auxiliary variables [19], is a prominent approach: it scales polynomially in the number of threads. However, the precision loss caused by multi-threaded Cartesian abstraction often leads to inconclusive results. In [23], we presented a method for refining multi-threaded Cartesian abstraction that is based on the so-called exception sets. An *exception set* is, roughly, a set of states that is excluded from the approximation of an abstraction. The refined method scaled in the number of threads. The refined method was also always able to return a conclusive answer, but the main ingredients were requested from the user. The user had to provide a parameter (namely, an exception set) which was not obvious to find.

In this paper we present an algorithm for finding and using exception sets automatically. The algorithm is complete, i.e. it can prove all properties of all programs. Moreover, the algorithm is polynomial not only for all programs that are provable by thread-modular reasoning, but also on a specific class of programs that are not amenable to thread-modular reasoning. We have implemented this

* Supported by DFG-Graduiertenkolleg GRK 806/2 "Mathematische Logik und Anwendungen" and by Transregional Collaborative Research Center 14 AVACS.

global $x = y = turn = 0$

A: $x := 1$;	A: $y := 1$;
B: $turn := 1$;	B: $turn := 0$;
C: **while**(y **and** $turn$);	C: **while**(x **and not** $turn$);
critical	**critical**
D: $x := 0$; **goto** A;	D: $y := 0$; **goto** A;

Fig. 1: Peterson's mutual exclusion algorithm.

algorithm and tested its performance. The evaluation indicates that the theoretical complexity guarantees can be realized in practice. In summary, the algorithm solves a new class of instances of the state explosion problem.

2 Illustration

In this section we provide a high-level illustration of our abstraction refinement algorithm. Using the Peterson's protocol [30] for mutual exclusion and a spurious counterexample produced by thread-modular verification approach we show the refinement computed by our procedure.

See Fig. 1 for a program implementing the protocol. We want to prove the mutually exclusive access to the location labeled D, i.e., that D is not simultaneously reachable by both processes.

The thread-modular approach interleaves the computation of reachable program states for each thread with the application of Cartesian abstraction among the computed sets. For our program the reachability computation traverses the following executions of the first and second thread, respectively (where each tuple represents a valuation of the program variables x, y, $turn$, pc_1, and pc_2):

$$(0,0,0,A,A),(0,1,0,A,B),(0,1,0,A,C),(0,1,0,A,D),(1,1,0,B,D)\ ,$$
$$(0,0,0,A,A),(1,0,0,B,A),(1,0,1,C,A),(1,1,1,C,B),(1,1,0,C,C)\ .$$

The last states of the executions above, i.e., the states $(1,1,0,B,D)$ and $(1,1,0,C,C)$, have equal valuations of the global variables and hence are subject to Cartesian abstraction, which weakens the relation between the valuations of local variables of individual threads. The application of Cartesian abstraction on this pair produces the following set of states:

$$\begin{aligned}\{(1,1,0)\} \times \{B,C\} \times \{D,C\} = \{&(1,1,0,B,D),(1,1,0,B,C),\\ &(1,1,0,C,D),(1,1,0,C,C)\}\ .\end{aligned}$$

The subsequent continuation of the reachability computation discovers that the first thread can reach an error state $(1,1,0,D,D)$ by making a step from the state $(1,1,0,C,D)$. That is, the thread-modular approach discovers a possibly spurious counterexample to the mutual exclusion property of our program.

The feasibility of the counterexample candidate is checked by a standard backwards traversal procedure starting from the reached error state $(1,1,0,D,D)$.

This check discovers that $(1,1,0,C,D)$ is the only state that can be reached backwards from $(1,1,0,D,D)$. That is, the counterexample is spurious and needs to be eliminated.

Now we apply our refinement procedure to refine the thread-modular approach. First, our procedure discovers that the application of Cartesian abstraction on the pair of states $(1,1,0,B,D)$ and $(1,1,0,C,C)$ produced the state $(1,1,0,C,D)$, since

$$(1,1,0,C,D) \in \{(1,1,0)\} \times \{B,C\} \times \{D,C\} \ ,$$

and identifies it as a reason for the discovery of the spurious counterexample. Second, the Cartesian abstraction used by the thread-modular approach is refined by adding $(1,1,0,B,D)$ (or, alternatively $(1,1,0,C,C)$) to the so-called *exception set* [23]. The states in the exception set are excluded from the Cartesian abstraction, thus refining it. As a result, the discovered spurious counterexample is eliminated since $(1,1,0,C,D)$ becomes unreachable. As in the existing counterexample guided abstraction refinement schemes, we proceed by applying the thread-modular approach, however now it is refined by the exception set $\{(1,1,0,B,D)\}$.

In addition to the above counterexample, the thread-modular approach also discovers a spurious counterexample that reaches the error state $(1,1,1,D,D)$. Our refinement procedure detects that the application of Cartesian abstraction on a state $(1,1,1,D,B)$ leads to this counterexample. Thus, the abstraction is refined by extending the exception set with the state $(1,1,1,D,B)$.

Finally, the thread-modular approach refined with the resulting exception set $\{(1,1,0,B,D),(1,1,1,D,B)\}$ proves the program correct. In Section 5, we present a detailed description of how our refinement method computes exception sets.

3 Preliminaries

Now we define multi-threaded programs, multi-threaded Cartesian abstraction and exception sets. We combat state space explosion in the number of threads, so we keep the internal structure of a thread unspecified.

An *n-threaded program* is given by sets Glob, Loc, $\rightarrow_i$ (for $1 \le i \le n$), init, where each $\rightarrow_i$ is a subset of $(\mathrm{Glob} \times \mathrm{Loc})^2$ (for $1 \le i \le n$) and init $\subseteq \mathrm{Glob} \times \mathrm{Loc}^n$.

The components of the multi-threaded program mean the following:

- The set of *shared states* Glob contains valuations of global variables
- The set of *local states* Loc contains valuations of local variables including the program counter (without loss of generality let all threads have equal sets of local states)
- $\rightarrow_i$ is the transition relation of the i^{th} thread ($1 \le i \le n$).
- init is the set of initial program states.

If the *program size* $|\mathrm{Glob}| + |\mathrm{Loc}| + \sum_{i=1}^{n} |{\rightarrow_i}| + |\mathrm{init}|$ is finite, the program is called *finite-state*.

The elements of States $=$ Glob $\times$ Loc^n are called *program states*, the elements of Glob $\times$ Loc are called *thread states*.

The program is equipped with the interleaving semantics: if a thread makes a step, then it may change its own local variables and the global variables but may not change the local variables of another thread; a step of the whole program is a step of some of the threads. The *post* operator maps a set of states to the set of their successors:

$$\begin{gathered}\mathrm{post}:\ 2^{\mathrm{States}} \to 2^{\mathrm{States}}\,,\\ S \ \mapsto\ \{(g',l') \mid \exists\,(g,l)\in S,\, i\in\mathbb{N}_n:\ (g,l_i)\to_i (g',l'_i) \text{ and } \forall\, j\neq i:\, l_j = l'_j\}\,,\end{gathered}$$

where $\mathbb{N}_n$ is the set of first n positive integers and the lower indices denote components of a vector. The verification goal is to show that any computation that starts in an initial state stays within the set of safe states, formally:

$$\bigcup_{k\geq 0} \mathrm{post}^k(\mathrm{init}) \subseteq \mathrm{safe}\,.$$

Thread-modular reasoning can prove the same properties as abstract fixpoint checking in the following setup [21, 22]:
$D = \mathfrak{P}(\mathrm{States})$ is the concrete domain, ordered by inclusion,
$D^\# = (\mathfrak{P}(\mathrm{Glob}\times\mathrm{Loc}))^n$ is the abstract domain, least upper bound $\sqcup$ is the componentwise union,

$$\begin{array}{lll}\alpha_{\mathrm{mc}} : D \to D^\# \,, & S \mapsto (\{(g,l_i) \mid (g,l)\in S\})_{i=1}^n \,,\\ \gamma_{\mathrm{mc}} : D^\# \to D \,, & T \mapsto \{(g,l) \mid \forall\, i\in\mathbb{N}_n : (g,l_i)\in T_i\}\,,\end{array}$$

are the abstraction and concretization maps which form the *multi-threaded Cartesian* Galois connection. Interestingly, the Owicki-Gries proof method without auxiliary variables [26] can prove exactly the same properties [19].

Given a set of states $E \subseteq$ States, the *exceptional* Galois connection

$$\begin{array}{ll}\alpha_E : D \to D, & S \mapsto S\setminus E\,,\\ \gamma_E : D \to D, & S\mapsto S\cup E\,.\end{array}$$

can be used to parameterize any abstract interpretation. In particular, the *parameterized multi-threaded Cartesian* Galois connection

$$(\alpha_{\mathrm{mc},E},\ \gamma_{\mathrm{mc},E}) \quad = \quad (\alpha_{\mathrm{mc}}\circ\alpha_E,\ \gamma_E\circ\gamma_{\mathrm{mc}})$$

allows arbitrarily precise polynomial-time analysis by a clever choice of the *exception set* E [23].

How to find a suitable exception set in acceptable time automatically? The remainder of the article deals with this question.

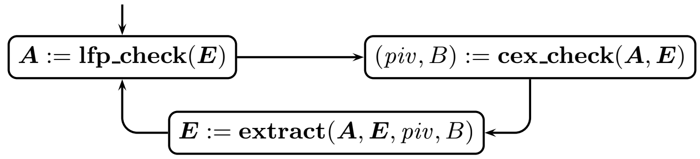

Fig. 2: TM-CEGAR: topmost level. The function **lfp_check** tries to compute an inductive invariant by generating the sequence $\boldsymbol{A}$ by abstract fixpoint iteration where $\boldsymbol{E}$ tunes the interpretation of elements of $\boldsymbol{A}$. In case an error state occurs in $\boldsymbol{A}$, the function **cex_check** determines the reason for the error occurrence. It determines the smallest iterate index piv such that the interpretation of A_{piv} has erroneous descendants later in $\boldsymbol{A}$. The function **extract** looks at the way A_{piv} was constructed, at those states in the concretization of this iterate that have erroneous successors, and tunes the parameters starting from E_{piv}.

4 Algorithm

Now we show TM-CEGAR, a thread-modular counterexample-guided abstraction refinement loop, that, given a multi-threaded program and a set of error states, proves or refutes nonreachability of error states from the initial ones.

The computation of TM-CEGAR on a particular multi-threaded program is a sequence of *refinement phases*, such that within each refinement phase, previously derived exception sets are used for the fixpoint iteration and a new exception set is computed. A refinement phase corresponds to one execution of the CEGAR loop.

TM-CEGAR operates on two sequences $\boldsymbol{A} = (A_i)_{i \geq 1} \in {D^{\#}}^{\omega}$ and $\boldsymbol{E} = (E_i)_{i \geq 1} \in D^{\omega}$. The sequence $\boldsymbol{A}$ is filled by the iterates of the abstract fixpoint computation, where each iterate A_i has a different interpretation which depends on E_i ($i \geq 1$).

The topmost level of TM-CEGAR is given in Fig. 2 (variables printed in bold face are sequences). Initially, the sequence of parameters $\boldsymbol{E}$ consists of empty sets. Let's fix a refinement phase, assuming that the sequence of parameters has already been constructed previously. Using parameters from $\boldsymbol{E}$, we construct the sequence of iterates $\boldsymbol{A}$ in the function **lfp_check**. Assuming abstract fixpoint computation has found error states in some iterate of $\boldsymbol{A}$, the function **cex_check** examines $\boldsymbol{A}$ to find the earliest states in $\boldsymbol{A}$ that are ancestors of the found error states. In case these earliest states don't contain initial ones, but lie in the interpretation of some iterate A_{piv}, the interpretations of A_{piv} and of all subsequent iterates are tuned by changing the parameters from E_{piv} onwards.

4.1 Abstract reachability analysis

The abstract fixpoint computation in Fig. 3 generates the sequence of iterates $\boldsymbol{A}$ based on the sequence of parameters $\boldsymbol{E}$.

The **lfp_check** function generates the first element of $\boldsymbol{A}$ as an abstraction of the initial states, parameterized by the first parameter: $A_1 = \alpha_{\mathrm{mc},E_1}(\mathrm{init})$. The

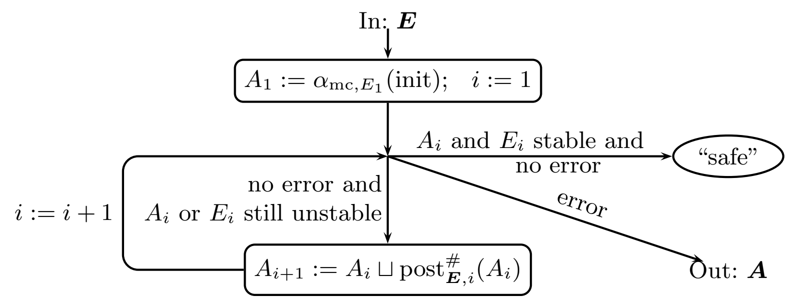

Fig. 3: The **lfp_check** function. The function $\mathrm{post}^{\#}_{\boldsymbol{E},i}$ is an abstract transformer whose precision is tuned by particular elements from $\boldsymbol{E}$. An error state is detected when $\gamma_{\mathrm{mc},E_i}(A_i) \not\subseteq$ safe. Stability of A_i and E_i means that $(E_{i-1}, A_{i-1}) = (E_i, A_i)$.

subsequent iterates are computed by taking the join of the current iterate with an approximation of post, applied to the current iterate. The approximation of post is tuned by $\boldsymbol{E}$:

$$\mathrm{post}^{\#}_{\boldsymbol{E},i} = \alpha_{\mathrm{mc},E_{i+1}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_i}\,.$$

The computation of iterates stops in two cases:

- Either the concretizations of the iterates remain in the safe states and no more grow, which happens when the sequences $\boldsymbol{A}$ and $\boldsymbol{E}$ get stable after some index;
- Or the concretization of some iterate contains an error state.

In the first case, TM-CEGAR has proven correctness of the program and thus exits immediately.

In the second case both sequences $\boldsymbol{A}$ and $\boldsymbol{E}$ are analyzed.

To optimize **lfp_check**, notice that the new and the old sequences of parameters share a common prefix (empty prefix in the worst case): say, $E_1 \dots E_j$ remained the same for some $j \geq 1$. Then $A_1 \dots A_j$ remain the same and don't have to be recomputed in the next refinement phase. This optimization doesn't have any influence on the asymptotic running time, but is a great help in practice.

4.2 Checking counterexample for spuriousness

The **cex_check** function assumes that error states are found in concretization of the iterate A_i and determines the earliest ancestors of those error states in $\boldsymbol{A}$.

To implement the high-level description of **cex_check** in Fig. 4, we compute the precise ancestors of the error states inside the concretizations of the iterates backwards. For that, we construct *bad regions* $\mathrm{Bad}_{piv}, \dots, \mathrm{Bad}_i$ as follows:

$$\mathrm{Bad}_i := \gamma_{\mathrm{mc},E_i}(A_i) \setminus \mathrm{safe}\,,$$
$$\mathrm{Bad}_{j-1} := \mathrm{pre}(\mathrm{Bad}_j) \cap \gamma_{\mathrm{mc},E_{j-1}}(A_{j-1}) \text{ for } j \leq i$$

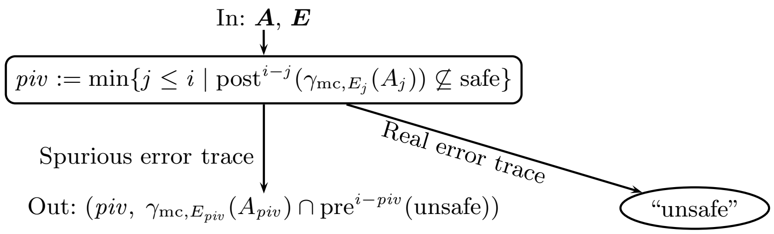

Fig. 4: The high-level view of the **cex_check** function. The set unsafe is States $\setminus$ safe. A real error trace is detected when $piv = 1 \wedge \mathrm{post}^{i-1}(\mathrm{init}) \not\subseteq \mathrm{safe}$. A spurious error is detected when $(piv = 1 \wedge \mathrm{post}^{i-1}(\mathrm{init}) \subseteq \mathrm{safe}) \ \vee \ piv > 1$.

until the bad region gets empty. The smallest iterate number *piv* for which the bad region is nonempty is called *pivot*. If pivot is 1 and there are initial states in Bad_1, the program has an error trace. Otherwise the error detection was due to the coarseness of the abstraction; another abstraction has to be chosen for the pivot iterate and subsequent iterates.

4.3 Refine: extract new exception set

Once a pivot iterate number *piv* is found, the exception set E_{piv} has to be enlarged to exclude more states from approximation. It is not obvious how to do that. We first specify requirements to the **extract** function, and then show the best known way of satisfying those requirements. Implementation variants of **extract** are discussed afterwards.

Requirements to extract. The precondition of **extract** is that $\emptyset \neq \mathrm{Bad}_{piv} \subseteq \gamma_{\mathrm{mc},E_{piv}}(A_{piv})$, and

- neither the interpretation of the previous iterate, namely, $\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})$,
- nor the successors of that interpretation

intersect Bad_{piv}. (Otherwise forward search would hit error states one iterate earlier or Bad_{piv-1} were nonempty.)

The postconditions imposed on the output of $\widetilde{\boldsymbol{E}}$ of **extract** are:

- $\gamma_{\mathrm{mc},\widetilde{E_{piv}}}(A_{piv-1} \sqcup \alpha_{\mathrm{mc},\widetilde{E_{piv}}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ doesn't intersect Bad_{piv} and
- $E_{piv} \subseteq \widetilde{E_{piv}} \subseteq E_{piv} \cup \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ and
- $\widetilde{E_k} = E_k$ for $k < piv$ and
- $\widetilde{E_k} = E_{piv}$ for $k > piv$.

The first postcondition ensures that no error trace starting at position *piv* and ending at position i would occur in **lfp_check** in the next refinement round. The second postcondition makes certain that previous spurious counterexamples starting at the pivot position would not reappear and that no new overapproximation is introduced. The third postcondition provides sufficient backtracking information for the next refinement phases. The last postcondition saves future computation time, intuitively, conveying the already derived knowledge to the future refinement phases; it may be relaxed, as we will see when discussing algorithm variants. The postconditions establish a monotonically growing sequence of parameters, and guarantee that the next sequence of interpretations of iterates is lexicographically smaller than the previous one, ensuring progress.

Implementation of extract. We gradually reduce the extraction problem to simpler subproblems.

First, we choose a set $\Delta E \subseteq \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ such that $\gamma_{\mathrm{mc},\Delta E}(A_{piv-1} \sqcup \alpha_{\mathrm{mc},\Delta E} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ doesn't intersect Bad_{piv}. Then we let $\widetilde{E_k} = E_k$ for $k < piv$ and $\widetilde{E_k} = \Delta E \cup E_{piv}$ for $k \geq piv$.

To choose such ΔE, we divide A_{piv-1}, $\mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ and Bad_{piv} into smaller elements of the abstract and concrete domains, such that the shared state within each small element is constant $g \in \mathrm{Glob}$:

$$
\begin{aligned}
A^{(g)} &= (\{(g,l) \in (A_{piv-1})_i\})_{i=1}^{n} \,, \\
P^{(g)} &= \{(g,l) \in \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))\} \,, \\
B^{(g)} &= \{(g,l) \in \mathrm{Bad}_{piv}\} \,.
\end{aligned}
$$

Then

$$
\begin{aligned}
A_{piv-1} &= \textstyle\bigsqcup_{g\in\mathrm{Glob}} A_{piv-1}^{(g)} \,, \\
\mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})) &= \textstyle\bigcup_{g\in\mathrm{Glob}} P^{(g)} \,, \\
\mathrm{Bad}_{piv} &= \textstyle\bigcup_{g\in\mathrm{Glob}} B^{(g)} \,.
\end{aligned}
$$

For each $g \in \mathrm{Glob}$, we have to find an exception set $\Delta^{(g)} \subseteq P^{(g)}$ such that $\gamma_{\mathrm{mc},\Delta E^{(g)}}(A_{piv-1}^{(g)} \sqcup \alpha_{\mathrm{mc},\Delta E^{(g)}}(P^{(g)}))$ doesn't intersect $B^{(g)}$. After having found such sets, we let $\Delta E = \bigcup_{g\in\mathrm{Glob}} \Delta E^{(g)}$.

Assume we have fixed $g \in \mathrm{Glob}$ and want to find $\Delta^{(g)}$ as above. To do that, it suffices to solve a similar problem for the standard Cartesian abstraction:

$$
\begin{aligned}
\alpha_c :&\ \ \mathfrak{P}(\mathrm{Loc}^n) \to (\mathfrak{P}(\mathrm{Loc}))^n \,, &\qquad S &\mapsto (\pi_i(S))_{i=1}^{n} \,, \\
\gamma_c :&\ (\mathfrak{P}(\mathrm{Loc}))^n \to \mathfrak{P}(\mathrm{Loc}^n) \quad , &\qquad ((T_i)_{i=1}^{n}) &\mapsto \textstyle\prod_{i=1}^{n} T_i \,,
\end{aligned}
$$

where π_i projects a set of tuples to the i^{th} component and index c means Cartesian. Namely, we are given a tuple $\tilde{A} \in (\mathfrak{P}(\mathrm{Loc}))^n$ and sets $\tilde{P}, \tilde{B} \subseteq \mathrm{Loc}^n$ such that $\tilde{B} \cap (\gamma_c(\tilde{A}) \cup \tilde{P}) = \emptyset$, and we want to find $\Delta\tilde{E} \subseteq \tilde{P}$ such that $\tilde{B} \cap \gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup (\tilde{P} \setminus \Delta\tilde{E})) = \emptyset$.

To solve this problem, we take the representation of $\tilde{B}$ as a union of products, say, $\tilde{B} = \bigcup_{j=1}^{m} \tilde{B}^{(j)}$ where $\tilde{B}^{(j)} = \prod_{i=1}^{n} \tilde{B}_i^{(j)}$ $(1 \leq j \leq m)$. Then we solve the problem for each $B^{(j)}$ instead of $\tilde{B}$ separately, and then take the union of the results.

So now let $j \in \mathbb{N}_m$ be fixed and let $\tilde{B} = \prod_{i=1}^{n} \tilde{B}_i$ be a Cartesian product such that $\tilde{B} \cap (\gamma_c(\tilde{A}) \cup \tilde{P}) = \emptyset$, as depicted on an example in Fig. 5a. We want to find $\Delta\tilde{E} \subseteq \tilde{P}$ such that $\tilde{B} \cap \gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup (\tilde{P} \setminus \Delta\tilde{E})) = \emptyset$. Since $\tilde{B}$ and $\gamma_c(\tilde{A})$ are products that don't intersect, there is a dimension $i \in \mathbb{N}_n$ such that $\tilde{B}_i$ and

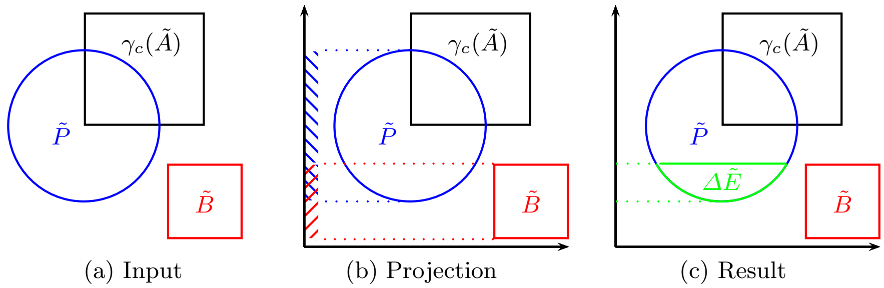

Fig. 5: Internals of $\mathbf{extract}(\boldsymbol{A}, \boldsymbol{E}, piv, B)$.

$\tilde{A}_i$ are disjoint (where $\tilde{A} = (\tilde{A}_i)_{i=1}^n$). In example on Fig. 5b, this is the vertical dimension $i = 2$. We let $\Delta\tilde{E} = \{p \in \tilde{P} \mid p_i \in B_i\}$, as in Fig. 5c.

Notice that $\tilde{P} \setminus \Delta\tilde{E}$ has no points whose i^{th} component occurs as the i^{th} component of a point of $\tilde{B}$. Thus the projections of two sets $\tilde{B}$ and of $\gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup \tilde{P} \setminus \Delta\tilde{E})$ onto the i^{th} component are disjoint. Thus the two sets are disjoint.

Variants of extract. Now we discuss another way of satisfying the stated postcondition of **extract** as well as a variant of those postconditions.

It turns out that taking not just one dimension in which $\tilde{B}_i$ and $\tilde{A}_i$ are disjoint, but all such dimensions (and solving the problem for each of the dimensions, and taking the union of the results), creates slightly larger sets on many examples, but saves future refinement phases in general. We call this variant of **extract** the *eager* variant. The total running time is decreased by a factor between 1 and 2, so we optionally use this variant in practice.

We may avoid more future refinement steps by creating exception sets not only for the iterate number piv, but also for as many iterate numbers between piv and i as possible, using, e.g., Bad_{piv} till Bad_i for B. This optimization requires a relaxed postcondition of **extract**. However, the effect of this optimization was insignificant on all the examples.

5 Applying TM-CEGAR to Peterson's protocol

In Section 2 we have sketched the main steps of TM-CEGAR on Peterson's protocol. Now we show the computation in more detail.

In the initial refinement phase, the sequence of exception sets $\boldsymbol{E}$ contains empty sets only. The procedure **lfp_check** starts with the iterate

$$A_1 \;=\; (\{(0,0,0,A)\},\ \{(0,0,0,A)\})\,,$$

where each tuple represents a valuation of program variables x, y, $turn$, pc. The **lfp_check** computation arrives at iterates (we skip A_2, A_3 as well as uninteresting states not having shared parts $(1,1,0)$ or $(1,1,1)$)

$$
\begin{aligned}
A_4 &= (\{(1,1,0)\} \times \{B\} \cup \{(1,1,1)\} \times \{C\} \cup \ldots, \\
&\quad \{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{B\} \cup \ldots), \\
A_5 &= (\{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{C,D\} \cup \ldots, \\
&\quad \{(1,1,0)\} \times \{B,C,D\} \cup \{(1,1,1)\} \times \{B,C\} \cup \ldots), \\
A_6 &= (\{(1,1,0)\} \times \{B,C,D\} \cup \{(1,1,1)\} \times \{C,D\} \cup \ldots, \\
&\quad \{(1,1,0)\} \times \{B,C,D\} \cup \{(1,1,1)\} \cup \{B,C,D\} \cup \ldots).
\end{aligned}
$$

The iterate A_6 is the earliest one whose concretization $\gamma_{\mathrm{mc},E_6}(A_6)$ contains error states, in this case $(1,1,0,D,D)$ and $(1,1,1,D,D)$. The forward computation detects those error states and hands $\boldsymbol{A}$ over to **cex_check**.

Notice that possible predecessors of the detected error states, namely, $(1,1,0,C,D)$ and $(1,1,1,D,C)$, are in the concretization of A_5. However, those states have no predecessors at all, thus the pivot iterate is 5. So **cex_check** returns $piv = 5$ and $B = \{(1,1,0,C,D),(1,1,1,D,C)\}$ and hands those values over to **extract**.

Procedure **extract** considers shared states $(1,1,0)$ and $(1,1,1)$ separately.

For shared state $(1,1,0)$, **cex_check** is given the tuple $\tilde{A} = (\{B\},\{B,C\})$ (obtained from A_4) and sets $\tilde{P} = \{B\} \times \{B,C,D\} \cup \{(C,C)\}$ (obtained from the successors of the concretization of A_4), $\tilde{B} = \{(C,D)\}$ (obtained from B). Notice that $\tilde{B}$ consists of one point only, which is trivially a product. Since $\tilde{A}_2 \cap \pi_2(\tilde{B}) = \{B,C\} \cap \{D\} = \emptyset$, **extract** can choose $\Delta\tilde{E} = \{p \in \tilde{P} \mid p_2 \in \pi_2(\tilde{B})\} = \{(B,D)\}$.

For shared state $(1,1,1)$, **extract** chooses $\Delta\tilde{E} = \{(D,B)\}$ analogously.

Thus the generated exception set is $\{(1,1,0,B,D),(1,1,1,D,B)\}$, which **extract** assigns to E_5, E_6, E_7, E_8, $\ldots$. The exceptions sets before the pivot, namely, E_1 till E_4, remain empty.

The next forward computation proceeds as the previous one till and including the iterate 4, and the iterate 5 is smaller than the previous one:

$$
\begin{aligned}
A_5 = (&\{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{C\} \cup \ldots, \\
&\{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{B,C\} \cup \ldots).
\end{aligned}
$$

The abstract fixpoint computation terminates at iterate 8 without finding an error state:

$$
\begin{aligned}
A_8 = (&\{(0,0,0,A),(0,0,1,A),(0,1,0,A),(0,1,1,A),(1,0,0,B)\} \\
&\cup \{(1,0,1),(1,1,0)\} \times \{B,C,D\} \cup \{(1,1,1)\} \times \{B,C\}, \\
&\{(0,0,0,A),(0,0,1,A)\} \cup \{(0,1,0)\} \times \{B,C,D\} \cup \{(0,1,1,B),(1,0,0,A)\} \\
&\cup \{(1,0,1,A)\} \cup \{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{B,C,D\})
\end{aligned}
$$

Its concretization $\gamma_{\mathrm{mc},E_8}(A_8)$ is an inductive invariant that contains no state of the form $(\_,\_,\_,D,D)$, so mutual exclusion is proven.

6 Parallel mutex loop

Now we will describe a practically interesting infinite class of programs. We show that TM-CEGAR can verify the programs of the class in polynomial time. We also show that our implementation can cope with the class better than the state-of-the-art tool SPIN.

$$
\textbf{bool}\ lck{=}0
$$

$$
\left[\begin{array}{ll}
 & \textbf{while}(\mathit{true})\ \{ \\
Q^0: & \textbf{acquire}\ lck; \\
R^{0,0}: & \\
\ldots & \textbf{critical} \\
R^{0,k-2}: & \\
R^{0,k-1}: & \textbf{release}\ lck; \\
Q^1: & \textbf{acquire}\ lck; \\
R^{1,0}: & \\
\ldots & \textbf{critical} \\
R^{1,k-2}: & \\
R^{1,k-1}: & \textbf{release}\ lck; \\
\vdots & \\
Q^{m-1}: & \textbf{acquire}\ lck; \\
R^{m-1,0}: & \\
\ldots & \textbf{critical} \\
R^{m-1,k-2}: & \\
R^{m-1,k-1}: & \textbf{release}\ lck; \\
 & \}
\end{array}\right]
\ \|\ \cdots\ \|\
\left[\begin{array}{ll}
 & \textbf{while}(\mathit{true})\ \{ \\
Q^0: & \textbf{acquire}\ lck; \\
R^{0,0}: & \\
\ldots & \textbf{critical} \\
R^{0,k-2}: & \\
R^{0,k-1}: & \textbf{release}\ lck; \\
Q^1: & \textbf{acquire}\ lck; \\
R^{1,0}: & \\
\ldots & \textbf{critical} \\
R^{1,k-2}: & \\
R^{1,k-1}: & \textbf{release}\ lck; \\
\vdots & \\
Q^{m-1}: & \textbf{acquire}\ lck; \\
R^{m-1,0}: & \\
\ldots & \textbf{critical} \\
R^{m-1,k-2}: & \\
R^{m-1,k-1}: & \textbf{release}\ lck; \\
 & \}
\end{array}\right]
$$

Fig. 6: Schema for programs consisting of n concurrent threads with m critical sections per thread such that each critical section has k control locations. The statement "**acquire** *lck*" waits until $lck = 0$ and sets it to 1. The statement "**release** *lck*" sets lck to 0. This class admits polynomial, efficient, precise and automatic thread-modular verification.

The most basic synchronization primitive, namely, a binary lock, is widely used in multi-threaded programming. For example, Mueller in Fig. 2 in [24] presents a C function that uses binary locks from the standard Pthreads library [18,25], through calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. The first function waits until the lock gets free and acquires it in the same transition, the second function releases the lock. Since we care about the state explosion problem in the number of threads only, we abstract away the shared data and replace the local operations by skip statements which change control flow location only.

Our class is defined by programs in Fig. 6. In a program of the class, each of n threads executes a big loop (a variant of the class in which threads have no loops but are otherwise the same has the same nice properties), inside of a loop a lock is acquired and released m times, allowing $k-1$ local operations inside each critical section. For instance, for the example of Mueller we have $k=3$, $m=1$, an unspecified n. This class extends the class presented in [23] by allowing variably long critical sections.

The property to be proven is mutual exclusion: no execution should end in a state in which some two threads are in their critical sections.

For a human, it might seem trivial that mutual exclusion holds. However, given just the transition relation of the threads, verification algorithms do have problems with the programs of the class. In fact, Flanagan and Qadeer [9] have shown a very simple program of this class that cannot be proven by thread-modular verification. Actually, it can be shown that no program of the class has a thread-modular proof.

6.1 Polynomial running time

Now we will show that our algorithm proves the correctness of mutex programs in polynomial time.

Theorem 1. *The running time of TM-CEGAR on a program from the mutex class is polynomial in the number of threads n, number of critical sections m and size of the critical section k.*

Proof. Let $C = \{R^{j,l} \mid j < m \text{ and } l < k\}$ be the critical local states, $N = \{Q^j \mid j < m\}$ the noncritical local states and $\mathrm{Loc} = C \dot\cup N$ the local states of a thread. A state (g, l) is an error state iff

$$\exists\, i, j \in \mathbb{N}_n : i \neq j \text{ and } a_i \in C \text{ and } a_j \in C\,.$$

For the proof of polynomial complexity we choose an eager version of **extract**, which is simpler to present. The eager version creates symmetrical exception sets for symmetrical inputs of **extract**: if interchanging two threads doesn't change the input of **extract**, the output is also stable under thread swapping.

The CEGAR algorithm needs $mk + 1$ refinement phases. In each phase, a new critical location is discovered. More specifically, in phases jk ($j < m$), the locations Q^j and $R^{j,0}$ are discovered. In phases $jk + r$ ($j < m$ and $1 \leq r < k$), the location $R^{j,r}$ is discovered. In each phase at least one new location is discovered because the set of error states has no predecessors and backtracking is not necessary. At the same time, no more than one critical location per phase is discovered: due to symmetry, when a new critical location of one thread is discovered, so it happens for all the threads. Since this critical location is new, it is not in the current exception set, thus it gets subjected to Cartesian abstraction, which leads to tuples with n critical locations (because of symmetry). Then the error states are hit and the exception set is enlarged. The eager version of **extract** produces, simplifying, all tuples where one component is critical and might include the new location (and the critical locations of the previous critical sections) and the other components are noncritical. This new exception set turns out to be equal to the current set of successors in their critical sections (if it were not, the difference between the successors and the exception set had at least one critical location, and, by symmetry, at least n, which would lead to error states after approximation). Subtracting the exception set from the successor set produces only tuples of noncritical locations, which get abstracted to a product of noncritical locations.

We just provide the central computation result, namely the exception set for each phase $jk + r$ (($j < m$ and $r < k$) or ($j = m$ and $r = 0$)); for details on intermediate exception sets, see [20]. We are interested in asymptotic behavior

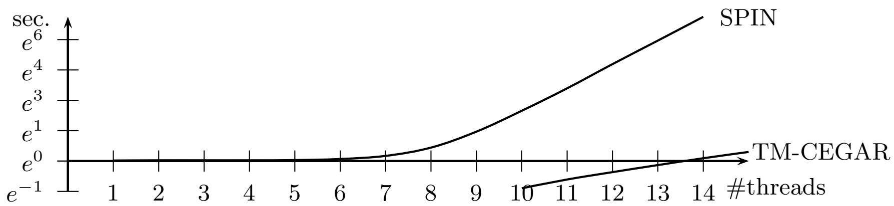

Fig. 7: SPIN vs. TM-CEGAR for 3 critical sections with one location each. Here $e \approx 2.7$ is the basis of natural logarithms. SPIN ran in exponential time $O(3.2^n)$, requiring 1892 seconds for 14 threads. TM-CEGAR needed only polynomial time $O(n^5)$, requiring around a second for 14 threads.

and thus show the derived exception set for large parameter values $n \geq 3$, $m \geq 1$ and $k \geq 2$ (for smaller values the exception sets are simpler).

Let $B(U, V)$ be the union over n-dimensional products in which exactly exactly one component set is V and the remaining are U. Now
$E_1 = \emptyset$ and
$E_{p(k+1)+2+l} = \{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid p' < p \wedge l' < k\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$, whose maximized form is
$\{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid (p' < p \wedge l' < k) \vee (p' \leq p \wedge l' \leq \min\{l, k-1\})\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$ for $p < j$ and $l \leq k$ as well as for $p = j$ and $l < r$,
the ultimate exception set is $E_{j(k+1)+1+r}$.

This representation is maximized: if some product is a subset of restriction of $E_{p(k+1)+2+l}$ to shared part 0 (resp. to shared part 1), it is a subset of some of the products in the above representation for shared part 0 (resp. for shared part 1).

Since each exception set has a polynomial-size maximized form, each refinement phase is polynomial-time by [23]. The number of refinement phases is also polynomial, so the total running time is polynomial. □

6.2 Experiments

We have implemented TM-CEGAR in OCAML and run tests on a 3MHz Intel machine.

We compared TM-CEGAR to the existing state-of-the-art tool SPIN 5.2.4 [13]. For comparison, we fixed $k = 1$ locations per critical section and $m = 3$ critical sections per thread, then we measured the running times of TM-CEGAR and SPIN in dependency on the number of threads n. We encoded the mutual exclusion property for SPIN by a variable which is incremented on acquires and decremented on releases, the property to be checked is that the value of this variable never exceeds one. The running times of SPIN and TM-CEGAR are depicted in Fig. 7 on a logarithmic scale.

SPIN fails at 15 threads, exceeding the 1 GB space bound, even if the most space-conserving switches are used (if default switches are used, SPIN runs out of

$m \setminus n$	1	10	20	30	40	50	60	70
1	0	0.30	6.94	46.72	185.51	553.47	1363.07	2912.25
3	0	10.90	235.36	1571.93	6085.92	17778.27	42848.10	90483.99
5	0	35.64	790.79	5260.06	20744.28	60610.99	147084.53	310593.29
7	0	80.29	1820.60	12276.20	48708.02	142755.15	346740.45	736117.10
9	0.01	150.52	3455.05	23538.67	94063,06	276296.15	671723.67	1432164.26

Fig. 8: Running times on the locks class for critical sections of size $k = 9$, a variable number of threads n and a variable number of critical sections m.

space for 12 threads already after 12 seconds). Compared to that, TM-CEGAR has a tiny running time, requiring around a second for 14 threads.

Fig. 8 demonstrates the behavior of TM-CEGAR on large examples. Of course, it is infeasible to wait for the completion of the algorithm on very large instances in practice. But we were astonished to see that TM-CEGAR requires negligibly small space. For example, after running for 3.4 days on the instance $n = 100$ threads, $m = 9$ critical sections of size $k = 1$, TM-CEGAR consumed at most 100MB; while after running for half a month on the instance $n = 80$, $k = m = 7$, it consumed only 150MB.

7 Related Work

The static analysis of multi-threaded programs has been and still is an active research topic [1–6, 8, 9, 11, 12, 14–17, 26–29]. Our work builds upon the thread-modular analysis to the verification of concurrent programs [9], which is based on an adaptation on the Owicki-Gries proof method [27] to finite-state systems.

In this paper we address the question of improving the precision of thread-modular analysis automatically, thus overcoming the inherent limitation of [9] to local proofs. We automate our previous work on exception sets [23] (which requires user interaction) by exploiting spurious counterexamples.

An alternative approach to improve the precision of thread-modular analysis introduces additional global variables that keep track of relations between valuations of local variables of individual threads [4]. As in our case, this approach is guided by spurious counterexamples. In contrast, our approach admits a complexity result on a specific class of programs for which the analysis is polynomial in the number of threads. Identifying a similar result for the technique in [4] is an open problem.

Keeping track of particular correlations between particular threads of a program can be dually seen as losing information about particular other threads [7,10]. Formally connecting CEGAR-TM with locality-based abstractions as well as the complexity analysis for the latter is an open problem.

Extensions for dealing with infinite-state systems (in rely-guarantee fashion) are based on counterexample-guided schemes for data abstraction [11]. While our method takes a finite-state program as input, we believe it can be combined with predicate abstraction over data to deal with infinite-state systems.

8 Conclusion

In this paper, we have presented the following contributions.

– An algorithm that takes the spurious counterexample produced by thread-modular abstraction and extracts the information needed for the subsequent refinement. The algorithm exploits the regularities of data structures for (unions of) Cartesian products and their operations.
– A *thread-modular counterexample-guided abstraction refinement* that automates the fine-tuning of an existing static analysis related to the thread-modular proof method. Previously, this fine-tuning was done manually.
– A static analysis method for multi-threaded programs that scales polynomially in the number of threads, for a specific class of programs. To the best of our knowledge, this is the first static analysis for which such a property is known, besides the thread-modular proof method which, however, can produce only *local proofs*.
– An implementation and an experimental evaluation indicating that the theoretical complexity guarantees can be realized in practice.

So far, we have concentrated on the state-explosion problem for multi-threaded programs, the concrete algorithms assume a finite-state program as an input. The assumption is justified if, e.g., one abstracts each thread. Doing so in a preliminary step may be too naive. Thus, an interesting topic for future work is the interleaving of thread-modular abstraction refinement with other abstraction refinement methods, here possibly building on the work of, e.g., [11, 12].

References

1. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551–, 2003.
2. S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.
3. E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI'06*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2005.
4. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 2007.
5. P. Cousot and R. Cousot. Invariance proof methods andanalysis techniques for parallel programs. In *Automatic Program Construction Techniques*, pages 243–271. Macmillan, 1984.
6. W.-P. de Roever. A compositional approach to concurrency and its applications. Manuscript, 2003.
7. J. Esparza, P. Ganty, and S. Schwoon. Locality-based abstractions. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 118–134. Springer, 2005.

8. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
9. C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.
10. P. Ganty. *The Fixpoint Checking Problem: An Abstraction Refinement Perspective*. PhD thesis, Université Libre de Bruxelles, 2007.
11. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In W. Pugh and C. Chambers, editors, *PLDI*, pages 1–13. ACM, 2004.
12. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In W. A. Hunt, Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.
13. G. J. Holzmann. The Spin model checker: Primer and reference manual. Addison-Wesley, ISBN 0-321-22862-6, `http://www.spinroot.com`.
14. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
15. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.
16. V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings in concurrent programs. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2009.
17. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2008.
18. X. Leroy. Pthreads linux manual pages. `https://www.sourceware.org/pthreads-win32/manual/pthread_mutex_init.html`.
19. A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2010.
20. A. Malkis and A. Podelski. Refinement with exceptions. Technical report, Albert-Ludwigs–Universität Freiburg, 2008. `http://wwwbroy.in.tum.de/~malkis/MalkisPodelski-refinementWithExceptions_techrep.pdf`.
21. A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification and Cartesian abstraction. Presentation at TV'06, 2006.
22. A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is Cartesian abstract interpretation. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2006.
23. A. Malkis, A. Podelski, and A. Rybalchenko. Precise thread-modular verification. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2007.
24. F. Mueller. Implementing POSIX threads under UNIX: Description of work in progress. In *Proceedings of the 2nd Software Engineering Research Forum*, Melbourne, Florida, Nov 1992.
25. B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., 1996.
26. S. S. Owicki. *Axiomatic Proof Techniques For Parallel Programs*. PhD thesis, Cornell University, Department of Computer Science, TR 75-251, July 1975.
27. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

28. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
29. S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI'2004*, pages 14–24. ACM, 2004.
30. A. U. Shankar. Peterson's mutual exclusion algorithm. `http://www.cs.umd.edu/~shankar/712-S03/mutex-peterson.ps`, note, 2003.