Precise Thread-Modular Verification

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

[1] University of Freiburg
[2] EPFL
[3] MPII

Abstract. Thread-modular verification is a promising approach for the verification of concurrent programs. Its high efficiency is achieved by abstracting the interaction between threads. The resulting polynomial complexity (in the number of threads) has its price: many interesting concurrent programs cannot be handled due to the imprecision of the abstraction. We propose a new abstraction algorithm for thread-modular verification that offers both high degree precision and polynomial complexity. Our algorithm is based on a new abstraction domain that combines Cartesian abstraction with *exception sets*, which allow one to handle particular thread interactions precisely. Our experimental results demonstrate the practical applicability of the algorithm.

1 Introduction

Many software systems are built from concurrent components. The development of such systems is a difficult and error prone task, since the programmer needs to write code that correctly handles all possible interactions between multiple concurrent threads. Verification of multi-threaded software is a hard problem [11]. The number of states of multi-threaded programs grows exponentially with the number of threads, which is called the state-explosion problem. There exist a variety of techniques and tools for the verification of multi-threaded programs, see e.g. [1, 2, 6–8, 14–16], which aim at reducing the number of states that needs to be inspected to verify a property.

One promising approach to circumvent the state explosion problem is offered by verification algorithms that reason about concurrent software modularly. Modularity allows one to avoid the explicit construction of the global state space by considering each thread in isolation, see e.g. [7, 9, 12]. The resulting polynomial complexity (in the number of threads) has its price: many interesting concurrent programs cannot be handled due to the imprecision of the abstraction [7]. For example, the existing thread-modular algorithms cannot prove the mutual exclusion property of the following simple concurrent fragment, which commonly appears in concurrent programs:

$$P_1 :: \begin{bmatrix} \ell_1 : \textbf{acquire}\ lck \\ \ell_2 : \textbf{critical} \\ \ell_3 : \textbf{release}\ lck \end{bmatrix} \quad \| \quad P_2 :: \begin{bmatrix} m_1 : \textbf{acquire}\ lck \\ m_2 : \textbf{critical} \\ m_3 : \textbf{release}\ lck \end{bmatrix}$$

Here, **acquire** lck waits until the lock variable lck becomes false and subsequently sets it to true. The call **release** lck sets the variable lck back to false. We observe

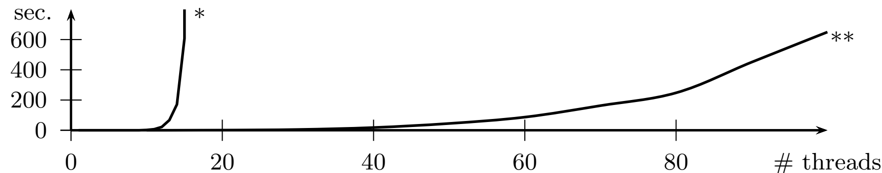


Fig. 1. Non-modular (∗) vs. thread-modular (∗∗) verification with exception set. We consider the example program given in Section 1 scaled w.r.t. the number of concurrent threads. Our algorithm retains polynomial complexity while gaining additional precision.

that the root of the imprecision lies in the fact that the thread-modular reasoning abstracts away crucial dependencies between local states of different threads, which are necessary to establish the property.

We propose a new abstraction algorithm for thread-modular verification that offers improved precision still within polynomial complexity. Our algorithm exploits the insight that we can prevent the undesired precision loss by preserving dependencies between certain sets of local states. These dependencies would otherwise be lost due to thread-modular abstraction. Stated in terms of abstraction, we exclude some a priori fixed set of program states from the abstraction process, and always treat them concretely. We refer to such sets as *exception sets*.

We formalize the notion of exception sets and their application in thread-modular verification in the framework of abstract interpretation [3], where we define a pair of abstraction and concretization functions that implement the application of exception sets. Now, we can combine any existing abstract interpretation with our exception set-based algorithm in a modular way, following [4]. In this paper, we study the combination of exception sets and Cartesian abstraction. Our interest in this combination is naturally motivated by the fact that thread-modular verification algorithms implement Cartesian abstraction [12]. We provide efficient algorithms for abstract interpretation in the combined abstraction, which retain the polynomial run time of the reachability computation with Cartesian abstraction while gaining precision from the exception sets. We identify an interesting class of concurrent programs for which our algorithm is precise and efficient. This class is obtained by parameterizing the fragment above with respect to the number of concurrent threads and the number of critical sections per thread.

We implemented our algorithm for precise thread-modular verification, and applied it on a series of benchmarks. The scalability of our implementation is promising: by using exception sets we were able to increase the number of concurrent threads that can be handled by our implementation by an order of magnitude, see Figure 1 and Section 6.

The main contributions of the paper consist of:

– an abstraction method with exception sets, which allows one to treat some part of the state space without abstraction;

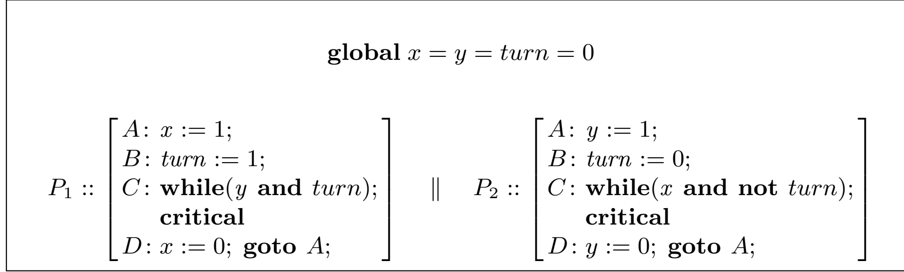


Fig. 2. Peterson's mutual exclusion algorithm.

- an implementation of the exception set-based abstraction with polynomial complexity in the number of threads and in the description of the exception set;
- an identification of a class of programs that allow verification in fully polynomial time;
- an experimental evaluation of a set of benchmarks that provides practical evidence for scalability of a prototype implementation.

The rest of the paper is organized as follows. First, we illustrate our approach for precise thread-modular verification with exception sets on a simple example. Section 3 formalizes abstraction and concretization with exception sets. In Section 4, we formally describe the verification algorithm. We present the class of programs on which our algorithm is precise and efficient in Section 5. Section 6 describes our experimental evaluation. We discuss the related work and conclude in Section 7. Some proofs are omitted due to the lack of space, and can be found in [13].

2 Example: Peterson's Algorithm

We illustrate our algorithm for the precise thread-modular verification on Peterson's mutual exclusion algorithm shown in Fig. 2. We wish to verify that at most one thread is in its critical section at location D.

First let us compute an over-approximation of the reachable states by applying a thread-modular verification algorithm, e.g. [7]. The result is represented by the following union of Cartesian products:

$$
\begin{array}{rlll}
 & \{000\} & \times\{A\} & \times\{A\} \\
\cup & \{001\} & \times\{A\} & \times\{A\} \\
\cup & \{010\} & \times\{A\} & \times\{A,B,C,D\} \\
\cup & \{011\} & \times\{A\} & \times\{A,B,C,D\} \\
\cup & \{100\} & \times\{B,C,D\} & \times\{A\} \\
\cup & \{101\} & \times\{B,C,D\} & \times\{A\} \\
\cup & \{110\} & \times\{B,C,D\} & \times\{A,B,C,D\} \\
\cup & \{111\} & \times\{B,C,D\} & \times\{A,B,C,D\}\,,
\end{array}
$$

where abc (e.g. 011) denotes the shared part $x = a \wedge y = b \wedge turn = c$ (e.g. $x = 0 \wedge y = 1 \wedge turn = 1$). This over-approximation is too coarse. It contains some states where both the first and the second thread are at their locations D, namely $(111, D, D)$ and $(110, D, D)$ (i.e. $x = y = turn = 1 \wedge pc_1 = pc_2 = D$ and $x = y = 1 \wedge turn = 0 \wedge pc_1 = pc_2 = D$).

Now we apply our algorithm instead. It iteratively computes an over-approximation of the reachable states, without losing the dependencies between the successors of the states contained in a given exception set. Let $E = \{(110, B, D), (110, C, C), (111, D, B)\}$ be the exception set.

We start the iteration with the initial state set $X_0 = \{(000, A, A)\}$. We compute the over-approximation of the set of states that are reachable from it in one step, as follows. First, we take the smallest Cartesian product that contains $(000, A, A)$. This is again

$$\{000\} \times \{A\} \times \{A\}\,.$$

Then we make a step which is specific to our algorithm. We extend this set by adding the elements of the exception set, which yields

$$X_1 = \{(000, A, A), (110, B, D), (110, C, C), (111, D, B)\}\,.$$

For this set, we compute the image under the one-step reachability under post, which is induced by the program, and add the initial element, which is in X_0. The resulting set is $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, C, C), (110, D, C), (111, C, D)\}$. Before over-approximating this set, we perform another step that is specific to our algorithm. We subtract the exception set from the result, which yields the set $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, D, C), (111, C, D)\}$. Then, for each shared part, we take the smallest Cartesian product that contains the local parts. This gives again the same set

$$\begin{aligned}
&\{000\} \times\{A\} \times\{A\} \\
\cup\, &\{010\} \times\{A\} \times\{B\} \\
\cup\, &\{011\} \times\{A\} \times\{B\} \\
\cup\, &\{100\} \times\{B\} \times\{A\} \\
\cup\, &\{110\} \times\{D\} \times\{C\} \\
\cup\, &\{111\} \times\{C\} \times\{D\}\,.
\end{aligned}$$

At last, we restore the states which get excluded before over-approximations, obtaining

$$\begin{aligned}
X_2 = \{&(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, B, D), \\
&(110, C, C), (110, D, C), (111, C, D), (111, D, B)\}\,.
\end{aligned}$$

We continue the fixpoint computation by applying the standard steps interleaved with the specific steps. The standard steps are taking one-step-successors and adding the initial states. The specific steps are subtracting the exception set away, applying the over-approximation and adding the exception set back.

The fixpoint of the described procedure is

$$
\begin{array}{lllll}
X_4 = & \{000\} & \times\{A\} & \times\{A\} \\
\cup & \{001\} & \times\{A\} & \times\{A\} \\
\cup & \{010\} & \times\{A\} & \times\{B,C,D\} \\
\cup & \{011\} & \times\{A\} & \times\{B\} \\
\cup & \{100\} & \times\{B\} & \times\{A\} \\
\cup & \{101\} & \times\{B,C,D\} & \times\{A\} \\
\cup & \{110\} & \times\{B,D\} & \times\{B,C\} \\
\cup & \{111\} & \times\{B,C\} & \times\{B,C,D\} \\
\cup & \multicolumn{3}{l}{\{(110,B,D),(110,C,C),(111,D,B)\}\,.}
\end{array}
$$

It is an inductive invariant of the program. Note that this over-approximation doesn't contain a state of the form $(___, D, D)$, so mutual exclusion is proven.

3 Abstraction with Exception

In this section, we formalize the notion of exception set in the framework of abstract interpretation [3]. In this setting, an exception set corresponds to an element E, called exception element, of the concrete domain D such that E is excluded from the abstraction. Additionally, we also exclude all concrete elements that are smaller than E from the abstraction, which follows the intuition that any subset of the exception set should also be excluded from the abstraction.

Let $(D, \subseteq)$ be a complete Boolean lattice and $(D^\#, \sqsubseteq)$ be a complete lattice. Let E be an exception element, and E^c be its complement. We define "exceptional abstraction" and "exceptional concretization" maps

$$\alpha_E : D \to D\ , \quad \alpha_E(X) = X \cap E^c\ ,$$

and

$$\gamma_E : D \to D\ , \quad \gamma_E(X) = X \cup E\ .$$

Proposition 1. *The pair (α_E, γ_E) is a Galois Connection. Formally:*

$$\forall\, X, Y \in D : \quad \alpha_E(X) \subseteq Y \Leftrightarrow X \subseteq \gamma_E(Y)\,.$$

Let (α, γ) be a Galois Connection with $\alpha : D \to D^\#$ and $\gamma : D^\# \to D$ such that γ maps the bottom of $D^\#$ to the bottom of D. The composition $(\alpha \circ \alpha_E, \gamma_E \circ \gamma)$ of the Galois Connections is again a Galois Connection. Let $\mathrm{init} \in D$ be any element, and F be a monotone function. We obtain an abstract interpretation algorithm that combines the abstraction (α, γ) with exception sets by computing the least fixpoint

$$\mathit{lfp}\ (\lambda Y.\, \alpha \circ \alpha_E(\mathrm{init} \cup F \circ \gamma_E \circ \gamma(Y))).$$

The concretization of this least fixpoint computed by applying $\gamma_E \circ \gamma$ over-approximates the least fixpoint of $\lambda x.\, \mathrm{init} \cup Fx$. Choosing E as its postfixed point (i.e. $\mathrm{init} \subseteq E$ and $FE \subseteq E$) makes this concretization equal to this postfixed fixpoint. So it is even possible to get exactly the least fixpoint of $\lambda x.\, \mathrm{init} \cup Fx$.

4 Precise Thread-Modular Verification

In this section, we formally present our method for thread-modular verification of multi-threaded programs, which uses exception sets for preserving dependencies between local states of different threads. We first describe multi-threaded programs. Then we provide necessary details on Cartesian abstraction, which is a basis for thread-modular verification. Finally, we describe how Cartesian abstraction can be efficiently combined with exception sets.

4.1 Multi-threaded Programs

A *multi-threaded program* is a tuple

$$(\mathrm{Glob}, \mathrm{Loc}, (\rightarrow_i)_{i=1}^n, \mathrm{init}),$$

where Glob and Loc are any sets, each $\rightarrow_i$ is a subset of $(\mathrm{Glob} \times \mathrm{Loc})^2$ (for $1 \leq i \leq n$) and $\mathrm{init} \subseteq \mathrm{Glob} \times \mathrm{Loc}^n$.
The meaning of different components of the multi-threaded program is the following:

- Loc contains valuations of local variables (including the program counter) of any thread, we call it the *local store* of the thread (without loss of generality we assume that all threads have equal local stores);
- Glob contains valuations of shared variables, we call it the *global store*;
- the elements of $\mathrm{States} = \mathrm{Glob} \times \mathrm{Loc}^n$ are called *program states*, the elements of $Q = \mathrm{Glob} \times \mathrm{Loc}$ are called *thread states*, the projection on the global store and the ith local store is the map

$$\pi_{\{0,i\}} : 2^{\mathrm{States}} \to 2^Q, \quad S \mapsto \{(g, l_i) \mid (g, l) \in S\}\,;$$

- the relation $\rightarrow_i$ is a transition relation of the ith thread ($1 \leq i \leq n$);
- init is a set of initial states.

The program is equipped with the usual interleaving semantics. This means that 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 successor operation maps a set of program states to the set of their successors:

$$\begin{aligned}&\mathrm{post} : 2^{\mathrm{States}} \to 2^{\mathrm{States}}\\ S \mapsto \{(g', l') \in \mathrm{States} \mid \exists\, (g, l) \in S,\, i \in \{1, ..., n\} : (g, l_i) \rightarrow_i (g', l'_i)\\ &\text{and } \forall\, j \neq i : l_j = l'_j\}.\end{aligned}$$

We are interested in proving safety properties of multi-threaded programs. Each safety property can be encoded as a reachability property and each reachability property can be encoded as reachability between a pair of states. So we are interested in whether there is a computation of any length $k \geq 0$ that starts in an initial state and ends in a single user-given error state f, formally:

$$\exists\, k \geq 0 : \; f \in \mathrm{post}^k(\mathrm{init})\,.$$

The state explosion problem in context of multi-threaded programs amounts to the fact that the number of program states is exponentially large in the number of threads n. We don't address the problem of growing state space due to the number of variables, which is also common to sequential programs.

4.2 Cartesian Abstract Interpretation

Thread-modular verification applies Cartesian abstraction to achieve polynomial complexity [12]. We briefly describe the necessary definitions below.

We present a concrete and an abstract domain and a Galois Connection between them that allows us to do abstract fixpoint checking. The definitions below extend the standard notion of the dependence-free abstraction [5]:

$D = 2^{\text{States}}$ is the set underlying the concrete lattice,
$D^{\#} = (2^{\text{Glob}\times\text{Loc}})^n$ is the set underlying the abstract lattice,
$\alpha_{\text{cart}} : \quad D \to D^{\#}$,
$\alpha_{\text{cart}}(S) = \big(\pi_{\{0,i\}} S\big)_{i=1}^n$
is the abstraction map, which projects a set of program states to the tuple of sets of thread states, so that the ith component of a tuple contains all states of the ith thread that occur in the set of program states.
$\gamma_{\text{cart}} : \quad D^{\#} \to D$,
$\gamma_{\text{cart}}(T) = \{(g, l) \mid \forall\, i \in \{1, ..., n\} : (g, l_i) \in T_i\}$
is the concretization map that combines a tuple of sets of thread states to a set of program of states by putting only those thread states together that have equal global part.

The ordering on the concrete domain D is inclusion, the least upper bound is the union $\cup$, the greatest lower bound is the intersection $\cap$, the complement X^c of a set X.

The ordering on the abstract domain $D^{\#}$ is the product ordering, i.e. $T \sqsubseteq T'$ if and only if $T_i \subseteq T'_i$ for all $i \in \{1, ..., n\}$. The least upper bound $\sqcup$ is componentwise union, the greatest lower bound $\sqcap$ is componentwise intersection. Thus the abstract lattice is complete. The bottom element is the tuple of empty sets $\bot = (\emptyset)_{i=1}^n$.

The pair of maps $(\alpha_{\text{cart}}, \gamma_{\text{cart}})$ is a Galois Connection, i.e. all $S \in D, T \in D^{\#}$ satisfy

$$\alpha_{\text{cart}}(S) \sqsubseteq T \text{ iff } S \subseteq \gamma_{\text{cart}}(T)\,.$$

4.3 Exception Set as Union of Maximal Cartesian Products

Our implementation of Cartesian abstraction combined with exception sets requires a suitable data structure for the representation of elements of the concrete and abstract domains. We analyze the representation of sets of tuples by sets of Cartesian products, which leads to a polynomial implementation, see Corollary 11.

We proceed by introducing some auxiliary propositions. Let D be any complete lattice with order $\leq$. Let us fix some "generating" subset of D so that

each element of D can be written as a join of some elements of the generating subset. Further let $Y \subseteq D$ be any set that contains the generating set so that the supremum of each chain in Y belongs to Y.

For $a \in D$, an element $y \in D$ is called *a-maximal*, if it is in Y, is less than or equal to a and there is no other greater element of Y that is less than or equal to a, formally:

$$y \in Y \text{ and } y \leq a \text{ and } \neg\exists\, y' \in Y : y < y' \leq a\,.$$

A set M is called *maximized*, if

$$M \subseteq Y \text{ and } (\forall\, y \in Y : y \leq \bigvee M \Rightarrow \exists\, y' \in M : y \leq y')\,.$$

Proposition 2. *Let $a \in D$. Then any element of Y less than or equal to a is less than or equal to some a-maximal element.*

Proposition 3. *Each element a of the lattice can be represented as a join of a unique maximized antichain. This maximized antichain contains exactly the a-maximal elements.*

Proposition 4. *Each maximized set contains the unique maximized antichain with the same join. Formally:*

$$\forall \text{ maximized } A \subseteq Y \;\; \exists^1 M \subseteq A : M \text{ is a maximized antichain and} \bigvee M = \bigvee A\,.$$

Now let us consider the Cartesian products. Recall that a function is a set of pairs so that for each first component there is exactly one second component. For an index set I, a *Cartesian product* of sets A_i $(i \in I)$ is the set of maps $\prod_{i\in I} A_i := \{f : I \to \cup_{i\in I} A_i \mid \forall\, i \in I : f(i) \in A_i\}$. For a subset of indices $J \subseteq I$ the *projection* of a subset $A \subseteq \prod_{i\in I} A_i$ on the components J is $\pi_J A = \{f : J \to \cup_{j\in J} A_j \mid \exists\, g \in A : f \subseteq g\}$. A projection on a single index $i \in I$ is $\pi_i A = \{a \in A_i \mid \exists\, g \in A : (i,a) \in g\}$. For a natural number n, the set $A^n := \prod_{i=1}^{n} A$ is the nth power of A.

Lemma 5. *Let A_i, B_i be sets indexed by $i \in I$. Then*

$$\prod_{i\in I} A_i \subseteq \prod_{i\in I} B_i \quad \Leftrightarrow \quad ((\forall\, i \in I : A_i \subseteq B_i) \;\text{ or }\; \exists\, i \in I : A_i = \emptyset)\,.$$

For the power set $D = 2^{(\mathrm{Loc}^n)}$ of all tuples of length n, ordered by inclusion, Y the set of all Cartesian products in D, and the set of singletons as a generating subset, the assumption is satisfied: singletons are Cartesian products and the union of a chain of Cartesian products is a Cartesian product.

By Proposition 3 every set of tuples can be represented as a union over a set of Cartesian products, so that no two Cartesian products from this set are comparable and this set is maximized. This is a crucial property for our representation of the exception set.

For a set of tuples $A \subseteq \mathrm{Loc}^n$, $i \in \mathbb{N}_n$ and $r \in \mathrm{Loc}$ let us call

$$A_{i,r} = \pi_{\mathbb{N}_n \setminus \{i\}} \{a \in A \mid a_i = r\}$$

a *restriction* of A (with parameters i, r). An $(n-1)$-tuple lies in this set exactly if, whenever r would be inserted at the ith position, the tuple would lie in A. Since projection is monotonic, restrictions are monotonic also, i.e.

$$\forall\, A \subseteq B \subseteq \mathrm{Loc}^n, i \in \mathbb{N}_n, r \in \mathrm{Loc} :\ A_{i,r} \subseteq B_{i,r}\,.$$

Lemma 6. *Let a set $A \subseteq \mathrm{Loc}^n$ be represented as a maximized antichain M of Cartesian products. Then for each $i \in \mathbb{N}_n, r \in \mathrm{Loc}$, the restriction $A_{i,r}$ has a representation as a union of a maximized antichain M' of Cartesian products with no greater cardinality than $|M|$. If* Loc *is finite and Cartesian products are stored componentwise, the elements of the new maximized antichain can be computed in polynomial time in n, $|\mathrm{Loc}|$ and $|M|$.*

In the following, we reduce the problem of computing the abstract parameterized post to a simpler problem about the "standard" Cartesian abstraction and concretization maps:

$$\alpha_c : 2^{(\mathrm{Loc}^n)} \to (2^{\mathrm{Loc}})^n, \quad \alpha_c(S) = (\pi_i S)_{i=1}^n\,,$$
$$\gamma_c : (2^{\mathrm{Loc}})^n \to 2^{(\mathrm{Loc}^n)}, \quad \gamma_c(T) = \prod_{i=1}^n T_i\,.$$

We call the elements of $(2^{\mathrm{Loc}})^n$ *Cartesian abstract* elements. The set of Cartesian products in Loc^n can be injected into the set of Cartesian abstract elements: a nonempty Cartesian product is bijectively mapped to the tuple of its components, the empty Cartesian product can be mapped to any tuple of sets among which at least one set is empty (for $n > 0$).

For the rest of this section we assume that the local store is finite. Each element of $(2^{\mathrm{Loc}})^n$ is represented as a list of n entries, each entry is itself a list of some elements from Loc.

Proposition 7. *The question whether a Cartesian product is a subset of a set of tuples can be solved in polynomial time.*

Formally: there is an algorithm that computes the map

$$2^{(\mathrm{Loc}^n)} \times (2^{\mathrm{Loc}})^n \to \mathrm{Bool},\ (E, A) \mapsto \gamma_c A \overset{?}{\subseteq} E$$

where E is represented as a set M of Cartesian abstract elements so that $\gamma_c M$ is a maximized antichain and $E = \bigcup \gamma_c M$, in polynomial time in $|M|$, n and $|\mathrm{Loc}|$.

Proposition 8. *The smallest Cartesian product that contains another Cartesian product without an exception set can be computed in polynomial time.*

Formally: there is an algorithm that computes the map

$$2^{(\mathrm{Loc}^n)} \times (2^{\mathrm{Loc}})^n \to (2^{\mathrm{Loc}})^n,\ (E, A) \mapsto \alpha_c(E^c \cap \gamma_c A)$$

where E is represented as a set M of Cartesian abstract elements so that $\gamma_c M$ is a maximized antichain and $E = \bigcup \gamma_c M$, in polynomial time in $|M|$, n and $|\mathrm{Loc}|$.

Proof. Let $E \subseteq \mathrm{Loc}^n$, $A \in (2^{\mathrm{Loc}})^n$. If $\gamma_c A$ is empty (which holds iff $A_i = \emptyset$ for some $i \in \mathbb{N}_n$), then the return value is the tuple of empty sets. Otherwise all A_i are nonempty.

Claim: All $r \in \mathrm{Loc}, i \in \mathbb{N}_n$ satisfy the equivalence:

$$r \in (\alpha_c(E^c \cap \gamma_c A))_i \quad \Leftrightarrow \quad r \in A_i \text{ and } \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j \not\subseteq E_{i,r}\,.$$

To prove the "$\Rightarrow$" direction, let $r \in (\alpha_c(E^c \cap \gamma_c A))_i = \pi_i(E^c \cap \gamma_c A)$. So there is an n-tuple $a \in E^c \cap \prod_{i=1}^n A_i$ with $a_i = r$, thus $r \in A_i$. Moreover $a \notin E$ and $a_j \in A_j$ ($j \in \mathbb{N}_n$). So the $(n-1)$-tuple $a \setminus \{(i,r)\} \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$, but $a \setminus \{(i,r)\} \notin E_{i,r}$.

To prove "$\Leftarrow$", let $r \in A_i$ and let a be an $(n-1)$-tuple with $a \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$ and $a \notin E_{i,r}$. Then $a \cup \{(i,r)\} \notin E$, but $a \cup \{(i,r)\} \in \prod_{j=1}^n A_j = \gamma_c A$. Thus $a \cup \{(i,r)\} \in E^c \cap \gamma_c A$, hence $r \in \pi_i(E^c \cap \gamma_c A) = (E^c \cap \gamma_c A)_i$.

The claim is proven. By Lemma 6, for each $i \in \mathbb{N}_n, r \in \mathrm{Loc}$, there is a maximized antichain $M'_{i,r}$ of Cartesian products with union $E_{i,r}$ and componentwise representation of Cartesian products as Cartesian abstract elements, computed in polynomial time. Since $M'_{i,r}$ is maximized, $A' \subseteq M'_{i,r}$ if and only if $\exists\, C \in M'_{i,r} :\ A' \subseteq C$ for any Cartesian product A', especially for $\prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j$. So all $r \in \mathrm{Loc}, i \in \mathbb{N}_n$ satisfy the equivalence:

$$r \in (\alpha_c(E^c \cap \gamma_c A))_i \qquad \Leftrightarrow \qquad r \in A_i \ \text{ and } \ \forall\, C \in M'_{i,r} : \prod_{j \in \mathbb{N}_n \setminus \{i\}} A_j \not\subseteq C\,.$$

Since $M'_{i,r}$ is generated in polynomial time and inclusion of Cartesian products is polynomial-time by Lemma 5, all the components of the abstract element $\alpha_c(E^c \cap \gamma_c A)$ are computable in polynomial time. □

Now we go over to the domains used in program analysis, namely to $D = 2^{\mathrm{States}} = 2^{\mathrm{Glob} \times \mathrm{Loc}^n}$ and $D^\# = (2^{\mathrm{Glob} \times \mathrm{Loc}})^n$.

Proposition 9. *The smallest abstract element that is greater than or equal to the concretization of another abstract element without an exception set can be computed in polynomial time.*

Formally: Assume that for $E \in D$, each $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) is represented a set of Cartesian abstract elements whose concretizations form a maximized antichain and have $\{l \mid (g,l) \in E\}$ (for $g \in \mathrm{Glob}$) as a union. Then computing the map

$$D \times D^\# \to D^\#, \quad (E, A) \mapsto \alpha_{\mathrm{cart}}(E^c \cap \gamma_{\mathrm{cart}} A)$$

needs polynomial time in n*,* $|\mathrm{Loc}|$*,* $|\mathrm{Glob}|$ *and the maximum cardinality of an antichain.*

Proof. Let $A \in D^{\#}$ and $E \in D$. For each $g \in \mathrm{Glob}$ and $i \in \mathbb{N}_n$ let $A_i^{[g]} := \{l \mid (g,l) \in A_i\}$ and $A^{[g]} := \prod_{i \in \mathbb{N}_n} A_i^{[g]}$. For all $g \in \mathrm{Glob}$, $l \in \mathrm{Loc}^n$ we have: $((g,l) \in \gamma_{\mathrm{cart}} A)$ iff $(\forall\, i \in \mathbb{N}_n : (g, l_i) \in A_i)$ iff $(\forall\, i \in \mathbb{N}_n : l_i \in A_i^{[g]})$ iff $((g,l) \in \{g\} \times \prod_{i=1}^n A_i^{[g]} = \{g\} \times A^{[g]})$. Thus

$$\gamma_{\mathrm{cart}} A = \bigcup_{g \in \mathrm{Glob}} \left(\{g\} \times A^{[g]} \right) . \tag{1}$$

For $g \in \mathrm{Glob}$, let $E^{(g)} = \{l \mid (g,l) \in E\}$. Any $g \in \mathrm{Glob}$ and $B \subseteq \mathrm{Loc}^n$ satisfy the equality:

$$(\{g\} \times B) \setminus E = \{g\} \times \left(B \setminus E^{(g)} \right) . \tag{2}$$

The map

$$\beta : \mathrm{Glob} \times \left(2^{\mathrm{Loc}}\right)^n \to D^{\#}, \quad (g, (B_i)_{i=1}^n) \mapsto (\{g\} \times B_i)_{i=1}^n$$

makes abstract elements from Cartesian abstract elements and is computable in polynomial time. Any $B \subseteq \mathrm{Loc}^n$ satisfies the equation:

$$\alpha_{\mathrm{cart}}(\{g\} \times B) = (\{(g, l_i) \mid l \in B\})_{i=1}^n = (\{g\} \times \pi_i B)_{i=1}^n = \beta\,(g, \alpha_c B) . \tag{3}$$

Now

$$\alpha_{\mathrm{cart}}\left(E^c \cap \gamma_{\mathrm{cart}} A\right) \overset{(1)}{=} \alpha_{\mathrm{cart}} \left(E^c \cap \bigcup_{g \in \mathrm{Glob}} \left(\{g\} \times A^{[g]} \right) \right) = [\text{distributivity}]$$

$$\alpha_{\mathrm{cart}} \left(\bigcup_{g \in \mathrm{Glob}} \left(\left(\{g\} \times A^{[g]} \right) \setminus E \right) \right) = [\text{abstraction map is a join-morphism}]$$

$$\bigsqcup_{g \in \mathrm{Glob}} \alpha_{\mathrm{cart}} \left(\left(\{g\} \times A^{[g]} \right) \setminus E \right) \overset{(2)}{=} \bigsqcup_{g \in \mathrm{Glob}} \alpha_{\mathrm{cart}} \left(\{g\} \times \left(A^{[g]} \setminus E^{(g)} \right) \right) \overset{(3)}{=}$$

$$\bigsqcup_{g \in \mathrm{Glob}} \beta \left(g, \alpha_c \left(\left(\gamma_c \left(A_i^{[g]} \right)_{i=1}^n \right) \setminus E^{(g)} \right) \right) .$$

From Prop. 8 we know that $\alpha_c \left(\left(\gamma_c \left(A_i^{[g]} \right)_{i=1}^n \right) \setminus E^{(g)} \right)$ is computable in polynomial time in n and $|\mathrm{Loc}|$ and the maximum cardinality of an antichain; the map β is also polynomial and the abstract join is also polynomial. □

Proposition 10. *Computing the best abstract post with exceptions takes polynomial time.*

Formally: Assume that for $E \in D$*, each* $\{l \mid (g,l) \in E\}$ *(for* $g \in \mathrm{Glob}$*) is represented as a set of Cartesian abstract elements whose concretizations form*

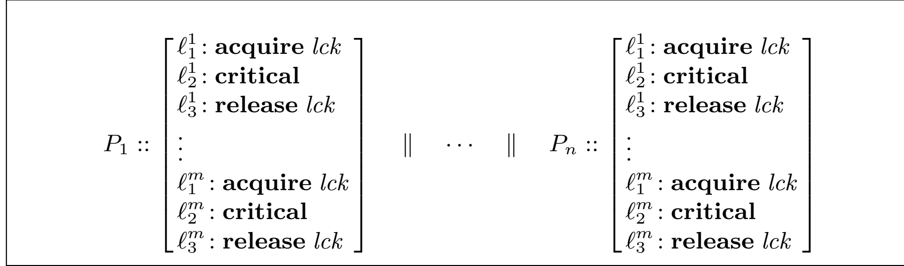


Fig. 3. Schema for programs consisting of n concurrent threads with m critical sections per thread, which admit efficient and precise thread-modular verification.

a maximized antichain and have $\{l \mid (g,l) \in E\}$ *as the union. Then computing the map*

$$D \times D^{\#} \to D^{\#}, (E, A) \mapsto \mathrm{post}_{E,\mathrm{cart}} A$$

takes polynomial time in n, $|\mathrm{Loc}|$, $|\mathrm{Glob}|$ *and the maximum size of an antichain from the representation of* E.

Corollary 11. *Computing the least abstract fixpoint with exceptional Cartesian abstraction and representation of* E *so that each* $\{l \mid (g,l) \in \mathrm{Loc}\}$ *is a union of Cartesian products needs polynomial time.*

Formally: Assume that for $E \in D$, *each* $\{l \mid (g,l) \in E\}$ *(for* $g \in \mathrm{Glob}$*) is represented as a set of Cartesian abstract elements whose concretizations form a maximized antichain and have* $\{l \mid (g,l) \in E\}$ *as the union. Then computing the map*

$$D \times D \to D^{\#}, \quad (E, \mathrm{init}) \mapsto \mathit{lfp}\ (\lambda X. \alpha_{\mathrm{cart}} \alpha_E (\mathrm{init} \cup \mathrm{post} \gamma_E \gamma_{\mathrm{cart}} X))$$

needs polynomial time in n, $|\mathrm{Loc}|$, $|\mathrm{Glob}|$, *in the cardinality of the largest antichain and in* $|\mathrm{init}|$.

Note that if initial states are represented the same way as the exception set then the run time is polynomial in the cardinality of the largest antichain from the representation of init instead of $|\mathrm{init}|$.

The whole algorithm can be viewed as a reduction to a polynomial number of queries of the form "is a Cartesian product a subset of a fixed set" as in Prop. 7. Each such query can be trivially answered given the representation of the fixed set as a union of all maximal (w.r.t. inclusion) Cartesian products inside this set.

5 Efficiently Handled Class

In this section, we describe a class of programs that can be efficiently verified by our thread-modular verification algorithm with exception sets.

Each program in the class is generated by instantiating the schema shown in Figure 3 with a fixed number n of threads and a fixed number m of critical sections per thread.

Let the sets of locations *InCrit* contain all local states at critical locations and *NotInCrit* be its complement:

$$
\begin{aligned}
\mathit{InCrit} &= \{l \in \mathrm{Loc} \mid \exists k : l(pc) \in \{\ell_2^k, \ell_3^k\}\} \\
\mathit{NotInCrit} &= \mathrm{Loc} \setminus \mathit{InCrit}
\end{aligned}
$$

Further for $1 \leq i \leq n$ let

$$
C(i) = \mathit{NotInCrit}^{i-1} \times \mathit{InCrit} \times \mathit{NotInCrit}^{n-i} \,,
$$

and

$$
M = \{C(i) \mid i \in \mathbb{N}_n\} \,.
$$

One can show that M is a maximized antichain. Now we choose

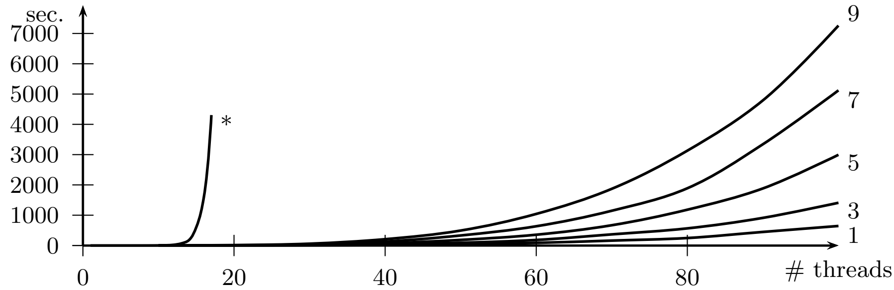
$$
E = \bigcup_{g \in \mathrm{Glob}, g(lck) \neq 0, C \in M} \{g\} \times C
$$

as an exception set. Checking the abstract fixpoint computed by parameterized thread-modular algorithm proves mutual exclusion. Moreover, all antichains in the representation of E have linear cardinality in n, so our algorithm consumes polynomial time and space. We conclude that no state explosion occurs during the application of our thread-modular algorithm.

6 Experiments

In this section we describe our experimental evaluation. We implemented the algorithm described in Section 4 in OCaml by using the ordered set data structure from the standard library to represent sets. We applied our implementation on a set of benchmark programs that we obtained by instantiating the schema shown in Figure 3. We experimented with the number of threads ranging from 10 to 100. For each thread size, we run our tool on programs with 1, 3, 5, 7, and 9 critical sections per thread. The resulting run times, which we obtained on 2.8 Ghz CPU, are shown in Figure 6.

We observe that our theoretical claims are supported by the experiments. The run time grows polynomially in the number of threads and in the number of critical sections. This allows us to verify instances of the program that are far beyond the reach of the algorithm that performs reachability computation without abstraction. Note that no existing thread-modular algorithm can handle the benchmark programs, due to the lack of precision.



	10	20	30	40	50	60	70	80	90	100
1	0.1	1.2	5.6	19	45	88	163	249	452	650
3	0.2	2.8	13	40	90	184	370	570	920	1423
5	0.3	5	26	72	186	359	675	1192	1901	3022
7	0.6	8.7	36	131	335	642	1158	1907	3374	5170
9	0.9	13.9	60	210	498	1052	1889	3160	4836	7328

Fig. 4. Experimental evaluation for the number of identical concurrent threads ranging between 10 and 100, and number of critical sections per thread from the set $\{1, 3, 5, 7, 9\}$. The table contains the run times, in seconds, for different combinations of number of critical sections/threads. The curve $*$ shows the run time for the exhaustive state exploration without abstraction for a single critical section (per thread), and puts the scale into perspective.

7 Related Work and Conclusion

Cartesian abstraction, which is also known as "independent attribute method", is a classical abstraction means in program analysis [10]. To the best of our knowledge, our application of Cartesian abstraction for the analysis of multi-threaded programs has not been known before.

The thread-modular verification algorithm of [7] serves as a starting point of our research, with the goals of improving its precision while retaining the polynomial complexity. The relationship between the thread-modular algorithm [7] and Cartesian abstraction provided a basis for the integration of exception sets into the abstraction framework.

In this paper, we presented a thread-modular verification algorithm that offers the polynomial complexity of the existing thread-modular approaches and increased precision. Such combination allows one to verify new classes of concurrent programs. Our experimental evaluation of the algorithm has shown its promising applicability.

The proposed algorithm is parameterized by an exception set, which determines the set of states that are excluded from the abstraction. We are currently developing an algorithm that computes an adequate exception set automatically. One possible direction is shown in Section 5.

8 Acknowledgements

The first author is supported in part by a DFG-Graduiertenkolleg of the University of Freiburg. The third author is supported in part by Microsoft Research through the European Fellowship Programme. We would like to thank anonymous reviewers for comments and suggestions.

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. E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI'2006*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
5. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181, 1995.
6. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL'2005*, pages 110–121. ACM, 2005.
7. 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.
8. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI'2004*, pages 1–13. ACM, 2004.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In W. A. H. Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.
10. N. D. Jones and S. S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *FOCS*, pages 185–190. IEEE, 1980.
11. D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.
12. 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 *LNCS*, pages 183–197. Springer, 2006.
13. A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification with arbitrary precision. `http://wwwbroy.in.tum.de/~malkis/MalkisPodelskiRybalchenko-threadModVerWithArbPrec.pdf`, technical report, 2006.
14. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'2005*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
15. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI'2004*, pages 14–24. ACM, 2004.
16. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003.