Abstract

Context-bounded analysis of concurrent programs is a technique to compute a sequence of under-approximations of all behaviors of the program. For a fixed bound $k$, a context bounded analysis considers only those runs in which a single process is interrupted at most $k$ times. As $k$ grows, we capture more and more behaviors of the program. Practically, context-bounding has been very effective as a bug-finding tool: many bugs can be found even with small bounds. Theoretically, context-bounded analysis is decidable for a large number of programming models for which verification problems are undecidable. In this paper, we survey some recent work in context-bounded analysis of multithreaded programs.

In particular, we show a general decidability result. We study context-bounded reachability in a language-theoretic setup. We fix a class of languages (satisfying some mild conditions) from which each thread is chosen. We show context-bounded safety and termination verification problems are decidable iff emptiness is decidable for the underlying class of languages and context-bounded boundedness is decidable iff finiteness is decidable for the underlying class.

1 Introduction

Algorithmic verification of shared-state multithreaded programs is one of the main motivations for research in theoretical computer science. The general problem is undecidable, even when the class of programs is restricted in different ways. Thus, one direction of research has focused on finding decidable models that over-approximate the problem and another on finding under-approximations. An over-approximate model captures more behaviors than the original program; thus, if we find that the over-approximation has no bad behaviors, we can be certain that neither does the original program. An under-approximation, conversely,
captures fewer behaviors. In this case, if we find a bad behavior in the approximation, we know that the bad behavior is also possible in the original program.

We consider a particular type of under-approximation: context bounding. Context-bounding is a technique to construct a parameterized sequence of under-approximations [46, 37]. For a fixed parameter \( k \), a \( k \)-context-bounded analysis considers only those behaviors of the program in which an individual thread is interrupted by the scheduler at most \( k \) times. As \( k \) increases, more and more behaviors of the original program fall into the purview of the analysis. In the limit, all behaviors are covered.

Context-bounding has become a popular technique because of two reasons. For a wide class of programming models and verification questions, context-bounded analyses become decidable, even though the unrestricted problems are undecidable. Moreover, in practice, context-bounded analysis has had success as a bug finding tool, since many bugs in practical instances can be discovered even with small values of \( k \) [46, 44, 36, 34].

We focus on decidability questions. In order to avoid “trivial” encodings of Turing machines, we restrict programs to be finite data—that is, we assume each program variable to take on finitely many values. Even with this restriction, depending on the model of programs, decidability can be non-immediate because the state space of a program can be infinite in other respects, such as the stack of an individual thread or the number of pending threads.

**Properties of concurrent programs**  For the moment, we focus on three decision problems: context-bounded reachability ("is there a \( k \)-bounded execution that reaches a specific global state?"), context-bounded termination ("all all \( k \)-bounded executions terminating?")], and context-bounded boundedness ("is there a bound on the number of pending threads along every \( k \)-bounded execution?"). We shall come back to other problems later.

Context-bounded analysis is a family of problems, depending on the model of concurrent programs as well as on the correctness properties considered. Qadeer and Rehof’s original paper [46], that introduced context-bounding, stipulated that there is a fixed number of recursive threads that read or write shared variables but these threads do not spawn further threads. They showed that the reachability problem is \( \text{NP} \)-complete. Note that even with two threads, the reachability problem for finite-data programs is already undecidable: for example, we can encode the intersection non-emptiness problem for pushdown automata. On the other hand, if threads are not recursive, then the reachability problem is decidable without any context bounding restrictions, even if threads can spawn further threads: this can be shown by a reduction to the coverability problem for vector addition systems with states (VASS). Subsequently, Atig, Bouajjani, and Qadeer [11] extended decidability for context-bounded reachability when threads can spawn further threads. They showed an upper bound of \( 2\text{EXPSPACE} \) and a matching lower bound was shown by Baumann et al. [14]. Similar techniques show the same complexity for termination and boundedness.

**A special case: Asynchronous programs**  The special case of \( k = 0 \) of context-bounded analysis is important enough to have its own name: asynchronous programs. In an asynchronous program, threads are executed atomically to completion (that is, never interrupted by the scheduler). Many software systems based on cooperative scheduling implement this model. Sen and Viswanathan [47] studied the model and showed reachability is decidable by reducing to a well-structured transition system. Ganty and Majumdar [28] showed that reachability, termination, and boundedness are all \( \text{EXPSPACE} \)-complete, by again reducing to coverability problems for VASS.

Majumdar, Thinniyam, and Zetzsche [40] proved decidability results for asynchronous
programs in a general language-theoretic setting. They fix a class of languages \( C \), and consider asynchronous programs in which each individual thread is a language from the class \( C \) over the alphabet of thread names as well as a transformer over the global states. That is, each thread is a language (from \( C \)) of words of the form \( dwd' \), where \( d \) and \( d' \) are global states and \( w \) is a sequence of thread names. The intent is that an atomic execution of the thread takes the global state from \( d \) to \( d' \) and also spawns new instances of all the threads in \( w \).

They show that for all classes \( C \) satisfying a mild language-theoretic assumption (the class \( C \) is a full trio), safety and termination are decidable if and only if the underlying language class \( C \) has a decidable emptiness problem. Similarly, boundedness is decidable if and only if finiteness is decidable for \( C \). As a consequence, they get decidability results for asynchronous programs over context-free languages, higher-order recursion schemes, as well as other language classes studied in infinite-state verification.

**Contribution**  Our starting point is the general approach of Majumdar, Thinniyam, and Zetzsche [40]. We show their general decidability results can be extended to context-bounded analysis (any \( k \geq 0 \)). We define concurrent programs over a language class \( C \) and show analogous decidability results: (i) context-bounded reachability and context-bounded termination for programs are decidable if and only if \( C \) has a decidable emptiness problem, and (ii) context-bounded boundedness is decidable if and only if \( C \) has a decidable finiteness problem. As a consequence, we get a uniform proof for decidability for these problems for programs over context-free languages and for programs over higher-order recursion schemes.

The key argument in both settings is that of downclosures of languages under the subword ordering. Safety, termination, and boundedness are preserved if we “lose” some spawned threads, as long as the sequence of global state changes (and there are at most \( k \) of them for the fixed context bound \( k \)) is maintained. Since downclosures (even when maintaining a bounded number of distinguished letters) are always regular languages, this implies: If our concurrent program satisfies one of the above properties, then each thread can be over-approximated by a regular language so that the property is still satisfied. The decision procedure for reachability then runs two semi-decision procedures: one enumerates executions (to check for reachability) and the other enumerates regular languages and checks that (1) the thread languages are contained in the regular languages and (2) uses known decidability results for context-bounded reachability with regular thread languages.

The decision procedure does not, in particular, need to construct an explicit description of the downclosure. In fact, it even shows decidability for language classes for which downclosures cannot be constructed. On the flip side, we do not get complexity bounds.

**Other properties**  What about other properties? Ganty and Majumdar showed fair termination for context-free asynchronous programs is decidable (by reduction to Petri net reachability) [28]. Majumdar, Thinniyam, and Zetzsche generalized the result to show that fair termination is equivalent to configuration reachability in the general setting [40]. On the other hand, decidability of fair termination implies the decidability of checking the “equal letters problem”: deciding if a language in \( C \) has an equal number of \( a \)s and \( b \)s. Thus, fair termination is undecidable for indexed languages. The undecidability is inherited by context-bounded fair termination. On the other hand, somewhat surprisingly, fair termination is decidable for context-bounded runs of context-free multithreaded programs [15].
2 Preliminaries

An alphabet is a finite non-empty set of symbols. For an alphabet Σ, we write Σ* for the set of finite sequences of symbols (also called words) over Σ. A set L ⊆ Σ* of words is a language. By \( \text{pref}(L) = \{ u \in \Sigma^* \mid \exists v \in \Sigma^*: wv \in L \} \) we denote the set of prefixes of words in L.

The subword order \( \subseteq \) on \( \Sigma^* \) is defined as follows: for \( u, v \in \Sigma^* \) we have \( u \subseteq v \) if and only if \( u \) can be obtained from \( v \) by deleting some of \( v \)'s letters. For example, \( abba \not\subseteq bababa \). The downclosure (or downward closure) \( \downarrow w \) of a word \( w \in \Sigma^* \) with respect to the subword order is defined as \( \downarrow w := \{ w' \in \Sigma^* \mid w' \subseteq w \} \). The downclosure \( \downarrow L \) of a language \( L \subseteq \Sigma^* \) is given by \( \downarrow L := \{ w' \in \Sigma^* \mid \exists w \in L: w' \subseteq w \} \). An important fact is that the subword ordering \( \subseteq \) is a well-quasi ordering (Higman’s lemma). A consequence is that the downclosure \( \downarrow L \) of any language \( L \) is a regular language [32]. However, a representation for the downclosure of a language may not be effectively constructible.

The projection of a word \( w \in \Sigma^* \) onto some alphabet \( \Gamma \subseteq \Sigma \), written \( \text{Proj}_\Gamma(w) \), is the word obtained by erasing from \( w \) each symbol which does not belong to \( \Gamma \). For a language \( L \), define \( \text{Proj}_\Gamma(L) = \{ \text{Proj}_\Gamma(w) \mid w \in L \} \). We write \( |w|_\Gamma \) for the number of occurrences of letters \( a \in \Gamma \) in \( w \), and similarly \( |w|_a \) if \( \Gamma = \{ a \} \).

A multiset \( m : X \to \mathbb{N} \) over a set \( X \) maps each symbol of \( X \) to a natural number. The size \( |m| \) of a multiset \( m \) is given by \( |m| = \sum_{x \in X} m(x) \). The set of all multisets over \( X \) is denoted \( \mathbb{M}[X] \). We identify subsets of \( X \) with multisets in \( \mathbb{M}[X] \) where each element is mapped to 0 or 1. We write \( m = [a, a, c] \) for the multiset \( m \in \mathbb{M}[\{a, b, c, d\}] \) such that \( m(a) = 2 \), \( m(b) = m(d) = 0 \), and \( m(c) = 1 \). The Parikh image \( \text{Parikh}(w) \in \mathbb{M}[\Sigma] \) of a word \( w \in \Sigma^* \) is the multiset such that for each letter \( a \in \Sigma \) we have \( \text{Parikh}(w)(a) = |w|_a \).

Given two multisets \( m, m' \in \mathbb{M}[X] \) we define \( m \oplus m' \in \mathbb{M}[X] \) to be the multiset such that for all \( a \in X \), we have \( (m \oplus m')(a) = m(a) + m'(a) \). If \( m(a) \geq m'(a) \) for all \( a \in X \), we also define \( m' \oplus m \in \mathbb{M}[X] \); for all \( a \in X \), we have \( (m' \oplus m)(a) = m(a) - m'(a) \). For \( X \subseteq Y \) we regard \( m \in \mathbb{M}[X] \) as a multiset in \( \mathbb{M}[Y] \) where undefined values are mapped to 0.

Language Classes and Full Trios A language class is a collection of languages, together with some finite representation. Examples are the regular languages (e.g. represented by finite automata) or the context-free languages (e.g. represented by pushdown automata). A relatively weak and reasonable assumption on a language class is that it is a full trio, that is, it is closed under rational transductions. Equivalently, a language class is a full trio if it is closed under each of the following operations: taking intersection with a regular language, taking homomorphic images, and taking inverse homomorphic images [16].

We assume that all full trios \( \mathcal{C} \) considered in this paper are effective: Given a language \( L \) from \( \mathcal{C} \), a regular language \( R \), and a homomorphism \( h \), we can compute a representation of the languages \( L \cap R \), \( h(L) \), and \( h^{-1}(L) \) in \( \mathcal{C} \).

Many classes of languages studied in formal language theory form effective full trios. These include the regular and the context-free languages [33], the indexed languages [2, 25], the languages of higher-order pushdown automata [42], higher-order recursion schemes [31, 24, 40], Petri nets [29, 35], and lossy channel systems. However, the class of deterministic context-free languages is not a full trio: this class is not closed under rational transductions.

3 A Language-Theoretic Model of Concurrent Programs

Intuitively, a concurrent program consists of a shared global state and a finite number of thread names. Instances of thread names are called threads. A configuration of such a
program consists of the current value of the global state and a multiset of partially-executed threads. A non-deterministic scheduler picks a partially-executed thread and runs it for some number of steps. An executing thread can change the global state. It can also spawn new threads—these can be picked and executed by the scheduler (in any order) in the future. When a scheduler swaps a running thread for another one, we say that there is a context switch. In our formal model, we keep the global state explicit and we model the execution behavior of threads as languages. The language of a thread captures the new threads it can spawn, as well as the effect of the execution on the global state.

3.1 Model

Let \( C \) be an (effective) full trio. A concurrent program (CP) over \( C \) is a tuple \( \mathfrak{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0) \), where \( D \) is a finite set of global states, \( \Sigma \) is an alphabet of thread names, \( (L_a)_{a \in \Sigma} \) is a family of languages from \( C \) over the alphabet \( \Sigma_D = D \cup \Sigma \cup (D \times D) \), \( d_0 \in D \) is an initial state, and \( m_0 \in M[\Sigma] \) is a multiset of initial pending thread instances. We assume that each \( L_a, a \in \Sigma, \) satisfies the condition \( L_a \subseteq aD(\Sigma \cup (D \times D))^{+} \) (we provide the intuition behind this condition below).

A configuration \( c = (d, m) \in D \times M[\Sigma_D^*] \) consists of a global state \( d \in D \) and a multiset \( m \) of strings representing pending threads instances and partially executed threads. Given a configuration \( c = (d, m) \), we write \( c.d \) and \( c.m \) to denote the elements \( d \) and \( m \), respectively.

The size of a configuration \( c \) is \( |c.m| \), i.e., the number of threads in the task buffer. We distinguish between threads that have been spawned but not executed (pending threads) and threads that have been partially executed (but swapped out). The pending thread instances are represented by single letters \( a \in \Sigma \) (which corresponds to the name of the thread) while the partially executed threads of “type” \( a \in \Sigma \) are represented by strings in \( \text{pref}(L_a) \) which end in a letter from \( D \times D \).

Before presenting the formal semantics, let us provide some intuition. Suppose the current configuration is \((d, m)\). A non-deterministic scheduler picks one of the outstanding threads (either a pending thread \( a \in m \) or a partially executed thread \( w \in m \) and executes it for some time until it terminates or until the scheduler decides to interrupt it. The execution of a thread \( a \) is abstractly modeled by the language \( L_a \). A word \( a_{d_1}w_1(d_{d_1})d_2w_2(d_{d_2})\ldots(d_{d_{k-1}})d_kw_{k+1}d_{k+1} \in L_a \) represents a run of an instance of the thread \( a \). The run starts executing in global state \( d_1 \). It spawns new threads \( w_1 \in \Sigma^* \), then gets interrupted at global state \( d_1' \) by the scheduler. At some future point, the scheduler starts executing it again at global state \( d_2 \), when new threads \( w_2 \) are spawned before it is interrupted again at \( d_2' \). The execution continues in this way until the thread terminates in global state \( d_{k+1} \). Thus, the jump from one global state to another (from the perspective of the thread) when a context switch is made is represented by a letter from \( D \times D \). The part of a run starting at global state \( d_i \), spawning threads \( w_i \) and interrupted at \( d_i' \) is called a segment. Each interruption is called a context switch; the above word has \( k \) context switches.

Formally, the semantics of \( \mathfrak{P} \) are given as a labelled transition system over the set of configurations with the transition relation \( \Rightarrow \subseteq (D \times M[\Sigma_D^*]) \times (D \times M[\Sigma_D^*]) \). The initial configuration is given by \( c_0 = (d_0, m_0) \).

The transition relation is defined using rules of four different types shown below. All four types of rules are of the general form \( d \xrightarrow{[w] \cdot m'} d' \). A rule of this form allows the program to move from a configuration \((d, m)\) to configuration \((d', m')\), i.e., \((d, m) \Rightarrow (d', m')\), iff \( d \xrightarrow{[w] \cdot m'} d' \) matches a rule and \((m \circ [w]) \oplus m' = m' \). Note that due to the definition of \( \circ \), \( m \) has to contain \( w \) for the rule to be applicable. We also write \( \xrightarrow{\alpha} \) to specify the particular
w used in the transition. As usual, the reflexive transitive closure of \( \Rightarrow \) is denoted by \( \Rightarrow^* \). A configuration \( c \) is said to be reachable if \( c_0 \Rightarrow^* c \).

(R1) \[ a \cdot \text{Parikh}(w) \circ [aw(d',d'')] \rightarrow d' \quad \text{if} \quad \exists w \in \Sigma^* : adw(d',d'') \in \text{pref}(L_a). \]

Rule (R1) allows us to pick some thread \( a \) from \( m \) and atomically execute it until the point it is switched out by the scheduler. Note that the final letter \( (d',d'') \) of the thread indicates that it has been switched out at global \( d' \) and can be resumed when the global state is \( d'' \).

(R2) \[ a \cdot \text{Parikh}(w) \circ d' \quad \text{if} \quad \exists w \in \Sigma^* : adw \in L_a. \]

Rule (R2) allows us to pick some thread \( a \) from \( m \) and atomically execute it to completion.

(R3) \[ aw'(d',d) \circ \text{Parikh}(w) \circ [aw'(d',d)w(d',d'')] \rightarrow d' \quad \text{if} \quad \exists w \in \Sigma^* : aw'(d',d)w(d',d'') \in \text{pref}(L_a). \]

Rule (R3) allows us to pick some partially executed thread and execute it atomically until the point it is switched out by the scheduler.

(R4) \[ aw'(d',d) \circ d' \quad \text{if} \quad \exists w \in \Sigma^* : aw'(d',d)w \in L_a \]

Rule (R4) allows us to pick some partially executed thread and execute it to completion.

### 3.2 Runs and Context-bounded Runs

A prerun of a concurrent program \( \mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0) \) is a finite or infinite sequence \( \rho = (e_0, n_0), w_1, (e_1, n_1), w_2, \ldots \) of alternating elements of configurations \( (e_i, n_i) \in D \times M[\Sigma_D^*] \) and strings \( w_i \in \Sigma^* \).

The set of preruns of \( \mathcal{P} \) will be denoted \( \text{Preruns}(\mathcal{P}) \). Note that if two concurrent programs \( \mathcal{P} \) and \( \mathcal{P}' \) have the same global states \( D \) and alphabet \( \Sigma \), then \( \text{Preruns}(\mathcal{P}) = \text{Preruns}(\mathcal{P}') \).

The length \( |\rho| \) of a finite prerun \( \rho \) is the number of configurations in \( \rho \).

A run of a CP \( \mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0) \) is a prerun \( \rho = (d_0, m_0), w_1, (d_1, m_1), w_2, \ldots \) starting with the initial configuration \( (d_0, m_0) \), where for each \( i \geq 0 \), we have \( (d_i, m_i) \xrightarrow{w_{i+1}} (d_{i+1}, m_{i+1}) \). The set of runs of \( \mathcal{P} \) is denoted \( \text{Runs}(\mathcal{P}) \). For a number \( k \), the run \( \rho \) is said to be \( k \)-context-bounded (\( k \)-CB for short) if for each \( c_i = (d_i, m_i) \in \rho \) and for each \( w \in m_i \), we have \( |w|_{D \times D} \leq k \). The set of \( k \)-context-bounded runs of \( \mathcal{P} \) is denoted by \( \text{Runs}_k(\mathcal{P}) \). In the case of finite runs which reach a certain configuration \( c \), we say a configuration \( c \) is \( k \)-reachable if there is a finite \( k \)-CB run \( \rho \) ending in \( c \).

### 3.3 Decision Problems

We study the following decision problems.

- **Definition 1.**
  - **CB Safety (Global state reachability):**
    
    **Instance:** A concurrent program \( \mathcal{P} \), a context-bound \( k \) and a global state \( d_f \in D \).
    
    **Question:** Is there a \( k \)-reachable configuration \( c \) such that \( c.d = d_f \)? If so, \( d_f \) is said to be \( k \)-reachable (in \( \mathcal{P} \)) and \( k \)-unreachable otherwise.

  - **CB Boundedness:**
    
    **Instance:** A concurrent program \( \mathcal{P} \) and a context-bound \( k \).
    
    **Question:** Is there an \( N \in \mathbb{N} \) such that for every \( k \)-reachable configuration \( c \) we have \( |c.m| \leq N \)? If so, the concurrent program \( \mathcal{P} \) is \( k \)-bounded; otherwise it is \( k \)-unbounded.

  - **CB Termination:**
    
    **Instance:** A concurrent program \( \mathcal{P} \), a context-bound \( k \).
    
    **Question:** Is \( \mathcal{P} \) \( k \)-terminating, that is, is every \( k \)-CB run of \( \mathcal{P} \) finite?
3.4 Orders on Runs and Downclosures

Intuitively, $k$-safety, $k$-termination, and $k$-boundedness are preserved when the multiset of pending threads is "$k$-lossy": pending threads can get lost and we only consider runs where each thread makes at most $k$ context switches. This loss corresponds to these pending threads never being scheduled by the scheduler. However, if a run demonstrates reachability of a global state, or non-termination, or unboundedness, in the $k$-lossy version, it corresponds also to a $k$-CB run in the original problem (and conversely). We make this intuition precise by introducing an ordering on runs and defining the downclosure.

Let $w, w' \in \Sigma^D \delta (\Sigma^D \times D)^* (D \cup (D \times D))$ be words with $w = a_d w_1 e_1 w_2 e_2 \ldots w_l e_l$ and $w' = a_d' w'_1 e'_1 w'_2 e'_2 \ldots w'_l e'_l$, where $a, a' \in \Sigma$, $d, d' \in D$, $e_1, e'_1 \in D \cup (D \times D)$, $w_i, w'_i \in \Sigma^*$ for $i, j \in [1, l]$ and $e_i, e'_i \in D \times D$ for $i, j \in [1, l - 1]$. We define the state-preserving order $\subseteq_D$ by $w \subseteq_D w'$ if $a = a'$, $d = d'$, $e_i = e'_i$ for each $i \in [1, l]$, and $w_i \subseteq w'_i$, that is, $w_i$ is a subword of $w'_i$ for each $i \in [1, l]$. We denote the corresponding notion of state-preserving downclosure under this order by $\downarrow$. Intuitively, the $\subseteq_D$ relation is a subword ordering on words that preserves the initial letter in $\Sigma$ and all occurrences of $D \cup (D \times D)$, but potentially loses letters from each segment—that is, newly spawned threads can be lost.

We use the order $\subseteq_D$ to naturally define the order $\preceq_D$ on $M[\Sigma^*_D]$ by induction: for $m, m' \in M[\Sigma^*_D]$ with $|m|, |m'| \geq 1$, we have $m \preceq_D m'$ if there are $n, n' \in M[\Sigma^*_D]$, $w, w' \in \Sigma_D^*$ with $m = n \uplus [w]$ and $m' = n' \uplus [w']$ such that $n \preceq_D n'$ and $w \subseteq_D w'$. Furthermore, for all $m \in M[\Sigma^*_D]$, we have $\emptyset \preceq_D m$.

We define an order $\leq$ on preruns as follows: For preruns $\rho = (e_0, n_0), w_1, (e_1, n_1), w_2, \ldots$ and $\rho' = (e'_0, n'_0), w'_1, (e'_1, n'_1), w'_2, \ldots$, we have $\rho \leq \rho'$ if $|\rho| = |\rho'|$, $e_i = e'_i$, $w_i \subseteq_D w'_i$ and $n_i \preceq_D n'_i$ for each $i \geq 0$. The downclosure $\downarrow R$ of a set $R$ of preruns of $\mathcal{P}$ is defined as $\downarrow R = \{ \rho \in \text{Preruns}(\mathcal{P}) \mid \exists \rho' \in R, \rho \leq \rho' \}$.

We write $\downarrow \text{Runs}(\mathcal{P})$ for the downclosure with respect to $\preceq$ restricted to valid runs.

Some properties of a concurrent program $\mathcal{P}$ only depend on the downclosure $\downarrow \text{Runs}_k(\mathcal{P})$ of the set $\text{Runs}_k(\mathcal{P})$ of $k$-CB runs of the program $\mathcal{P}$. For these properties, we may transform the program $\mathcal{P}$ to a program $\downarrow \text{Runs}_k(\mathcal{P})$ such that the latter is easier to analyze but retains the properties of the former.

Definition 2. For a language $L_a$ of a CP, let

$$\downarrow \text{Runs}_k(\mathcal{P}) = \downarrow \text{Preruns}_k(\mathcal{P})$$

For any CP $\mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0)$ and number $k$, we define the CP $\downarrow \text{Runs}_k(\mathcal{P}) = (D, \Sigma, (\downarrow \text{Runs}_k(\mathcal{P})))_{a \in \Sigma}, d_0, m_0)$. In other words, $\downarrow \text{Runs}_k(\mathcal{P})$ is the program obtained by taking the state-preserving downclosure of those words in $L_a$ which contain at most $k$ context switches.

Note that, by well-quasi-ordering arguments, for any fixed $k$, the languages $L_a$ of $\downarrow \text{Runs}_k(\mathcal{P})$ are all regular.

Proposition 3. Let $\mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0)$ be a concurrent program. Then $\downarrow \text{Runs}_k(\mathcal{P}) = \downarrow \text{Runs}(\downarrow \text{Runs}_k(\mathcal{P}))$. In particular,

1. For every $d \in D$, $\mathcal{P}$ can $k$-reach $d$ if and only if $\downarrow \text{Runs}_k(\mathcal{P})$ can $k$-reach $d$.
2. $\mathcal{P}$ is $k$-terminating if and only if $\downarrow \text{Runs}_k(\mathcal{P})$ is $k$-terminating.
3. $\mathcal{P}$ is $k$-bounded if and only if $\downarrow \text{Runs}_k(\mathcal{P})$ is $k$-bounded.

Clearly, every run in $\text{Runs}_k(\mathcal{P})$ is also in $\text{Runs}(\downarrow \text{Runs}_k(\mathcal{P}))$. Conversely, we can show by induction on the length of the run that for every run $\rho \in \text{Runs}(\downarrow \text{Runs}_k(\mathcal{P}))$ there is a run $\rho' \in \text{Runs}(\mathcal{P})$ such that $\rho \leq \rho'$. The result follows.
4 Decidability Results

We now characterize full trios \( C \) for which decision problems for concurrent programs over \( C \) are decidable. We shall make use of the following decidability results about regular languages.

▶ **Theorem 4.** 1. [28, 10] CB Safety is decidable for concurrent programs over regular languages.
2. [28, 15] CB Boundedness and CB termination are decidable for concurrent programs over regular languages.

In fact, the above problems are decidable even if there is no bound on the number of context switches. The result in [10] is stated for a model called Dynamic networks of Concurrent Finite-state Systems (DCFS), but it is easy to see that there is a polynomial time reduction for the problems of safety, termination and boundedness for CP over regular languages to the corresponding problems for DCFS. The paper [15] shows decidability of CB termination and CB boundedness for the model of dynamic networks of concurrent pushdown systems, of which DCFS is a special case. There is also a simple reduction of these problems to the corresponding results for the model of asynchronous programs [28].

Our first decidability result is the following.

▶ **Theorem 5.** Let \( C \) be a full trio. The following are equivalent:
(i) CB Safety is decidable for concurrent programs over \( C \).
(ii) CB Termination is decidable for concurrent programs over \( C \).
(iii) Emptiness is decidable for \( C \).

The implications “(i)⇒(iii)” and The implications “(ii)⇒(iii)” are immediate from corresponding results for asynchronous programs [40], since context bounded analysis problems generalize the corresponding analysis for asynchronous programs.

Before we prove the next implication, let us introduce a bit of notation. For each \( i \in \mathbb{N} \), let \( R_i \) be the regular language \( R_i = \Sigma D \Sigma^*(((D \times D) \Sigma^*)^i D) \), \( R'_i = \Sigma D \Sigma^*(((D \times D) \Sigma^*)^i (D \times D)) \) for each \( l \in \mathbb{N} \) we define \( R_l = \bigcup_{i=0}^{l} (R_i \cup R'_i) \). For any language \( L \) and \( k \in \mathbb{N} \), the language \( L \cap R_k \) captures those words in \( L \) that contain at most \( k \) context switches.

For the implication “(iii)⇒(i)”, we construct two semi-decision procedures (Algorithm 1): the first one searches for regular over-approximations \( A_a \) of each language \( L_a \) such that the program \( \Psi' \) obtained by replacing each \( L_a \) by the corresponding \( A_a \) is safe. We can check whether our current guess for \( \Psi' \) is safe using Theorem 4. By Proposition 3, we know that in case \( \Psi \) is safe, then there must exist such a safe regular over-approximation. Concurrently, the second procedure searches for a \( k \)-CB run reaching the target global state \( d \) which witnesses the negation. Clearly, one of the two procedures must terminate. Note that we use an emptiness check to ensure that our current guess for \( A_a \) includes the set \( L_a \cap R_k \).

To show “(iii)⇒(ii)”, we need an algorithm for termination of concurrent programs. As in the case of safety, it consists of two semi-decision procedures. The one for termination works just like the one for safety: It enumerates regular over-approximations and checks if one of them terminates. The procedure for non-termination requires some terminology:

**Predictions** We will use a notion of prediction, which assigns to each configuration \((e, n)\) of a run a multiset of strings that encode not only the past of each thread (as is done in \( n \)), but also its future. To do this, we define the alphabet \( \Gamma_D = \Sigma_D \cup \{\#\} \) that extends \( \Sigma_D \) a fresh letter \#. We shall encode predictions using strings of the form \( au\#v \), which encode a thread with name \( a \), past execution \( au \), and future execution \( v \). Additionally, we extend the
Algorithm 1 Checking CB Safety

Input: Concurrent program \( \mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0) \) over \( \mathcal{C} \), context bound \( k \in \mathbb{N} \), state \( d \in D \)

run concurrently

\[
\begin{array}{l}
\text{begin} \\
\text{// find a safe over-approximation */} \\
\text{foreach tuple} \ (A_a)_{a \in \Sigma} \ \text{of regular languages} \ A_a \subseteq \Sigma^* \ \text{do} \\
\text{if} \ \ (L_a \cap R_a) \cap (\Sigma_D^* \setminus A_a) = \emptyset \ \text{for each} \ a \in \Sigma \ \text{then} \\
\quad \text{return} \ d \ \text{is not reachable.} \\
\text{begin} \\
\text{// find a run reaching} \ d \ \text{*/} \\
\text{foreach prerun} \ \rho \ \text{of} \ \mathcal{P} \ \text{do} \\
\quad \text{if} \ \rho \ \text{is a} \ k\text{-CB run that reaches} \ d \ \text{then} \\
\quad \text{return} \ d \ \text{reachable.}
\end{array}
\]

order \( \preceq_D \) to strings of the form \( au#v \) by treating \# as a letter from \( D \times D \) which is to be preserved. Let us make this precise.

Suppose \( \rho \) is a (finite or infinite) prerun \((e_0, n_0), (e_1, n_1), \ldots \). An annotation for \( \rho \) is a sequence \( f_0, f_1, \ldots \in M[\Gamma^*_D] \) of multisets of strings such that the sequence has the same length as \( \rho \). If \( \rho \) is a run, then we say that the annotation \( f_0, f_1, \ldots \) is a prediction if

1. each string occurring in \( f_0, f_1, \ldots \) is of the form \( au#v \) such that \( auw \in \Sigma_D^* \) and \( auv \in L_a \)
2. for each \( i \geq 0 \), the multisets \( n_i \) and \( f_i \) have the same cardinality and there is a bijection between \( n_i \) and \( f_i \), so that (i) each word \( au \) in \( n_i \) is in bijection with some word \( au#v \) in \( f_i \), and (ii) if \( au \) is the active thread when going from \((e_i, n_i)\) to \((e_{i+1}, n_{i+1})\) and \( au#v \) is its corresponding string \( au#v \) in \( f_i \), then the system executes the next segment in \( v \).

Note that then indeed, for each thread, its string in \( n_i \) records its past spawns, whereas the corresponding string in \( f_i \) contains all its future spawns (and possibly an additional suffix).

Of course, for each (finite or infinite) run, there exists a prediction: Just take the sequence of actions of each thread in the future. Moreover, taking a prefix of both a run and some accompanying prediction will yield a (shorter) run with a shorter prediction.

Self-covering runs Recall that for each alphabet \( \Theta \), we have an embedding relation \( \preceq_D \) on the set \( M[\Theta_D^*] \), and in particular on \( M[\Gamma_D^*] \). We say that a finite run \((e_0, n_0), (e_1, n_1), \ldots, (e_m, n_m)\) together with a prediction \( f_0, f_1, \ldots, f_m \) is \( k\)-self-covering if for some \( i < m \), we have \( e_i = e_m, f_i \preceq_D f_m \), and also, all words in \( n_0, n_1, \ldots \) contain at most \( k \) context-switches. As the name suggests, self-covering runs are witnesses for non-termination:

Lemma 6. For every \( k \in \mathbb{N} \), a concurrent program has an infinite \( k\)-CB run if and only if it has a \( k\)-self-covering run.

Here, it is crucial that for each \( k \in \mathbb{N} \), the ordering \( \preceq_D \) is a WQO on the set of words with at most \( k \) context-switches (on all of \( \Sigma_D^* \), \( \preceq_D \) is not a WQO).

We can now decide termination (Algorithm 2): the algorithm either (i) exhibits a \( k\)-self-covering run, which shows the existence of a \( k\)-bounded infinite run by Lemma 6, or (ii) finds a regular over-approximation that terminates, which means the original program is terminating. We can check termination of the regular over-approximation using Theorem 4. The algorithm also terminates: If there is an infinite \( k\)-bounded run, then Lemma 6 yields the existence of a \( k\)-self-covering run. Moreover, if the concurrent program does terminate,
Algorithm 2 Checking CB Termination

Input: Concurrent program $\Psi = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0)$ over $C$ and context bound $k \in \mathbb{N}$

begin
  /* find a terminating over-approximation */
  foreach tuple $(A_a)_{a \in \Sigma}$ of regular languages $A_a \subseteq \Sigma_D^*$ do
    if $(L_a \cap R_k) \cap (\Sigma^* \setminus A_a) = \emptyset$ for each $a \in \Sigma$
      then return $\Psi$ is $k$-terminating.

begin
  /* find a self-covering run */
  foreach $\rho$ of $\Psi$ and an annotation $\sigma$ do
    if $\rho$ with $\sigma$ is a $k$-self-covering run
      then return $\Psi$ is not $k$-terminating.

then Proposition 3 ensures the existence of a terminating regular over-approximation. This concludes our proof of Theorem 5.

Our second theorem is as follows.

Theorem 7. Let $C$ be a full trio. The following are equivalent:

(i) CB Boundedness is decidable for concurrent programs over $C$.

(ii) Finiteness is decidable for $C$.

The implication “(i)$\Rightarrow$(ii)” follows from the special case of asynchronous programs [40]. It was also observed in [40] that decidability of finiteness for $C$ implies decidability of emptiness for $C$. Further, by Theorem 5, we may assume that CB safety is decidable for CP over $C$.

We now show the implication “(ii)$\Rightarrow$(i)”. For a language $L \subseteq \Sigma_D^*$ and $n \in \mathbb{N}$, let $L|_n = L \cap \Sigma_D^{\leq n}$ be the language restricted to strings of length at most $n$ and, in addition, for $k \in \mathbb{N}$, let $L|_n = L|_n \cap R_k$. Moreover, for an alphabet $\Theta$, a language $L \subseteq \Theta^*$, and a word $w \in \Theta^*$, we define the left quotient of $L$ by $w$ as $w^{-1}L := \{ u \in \Theta^* \mid \text{wu} \in L \}$. Our algorithm is based on the following characterization of unboundedness.

Lemma 8. The program $\Psi$ is $k$-unbounded iff one of the two following conditions hold:

(P1) Either there exists some number $n$ such that $\Psi_n = (D, \Sigma, (L_a|_n)_{a \in \Sigma}, d_0, m_0)$ is unbounded, or

(P2) for some $a \in \Sigma$, there exists some word $w \in \text{pref}(L_a)$ ending in a letter $(d, d') \in D \times D$ such that $\text{pref}(w^{-1}L_a) \cap \Sigma^*$ is infinite and there exists a run $\rho$ reaching a configuration $c$ with $w \in c$ and $c.d = d'$.

Essentially, (P1) captures the case where each thread spawns a finite number of other threads and (P2) the case where there is some reachable configuration at which a single thread can spawn an unbounded number of new threads. The above characterization allows us to implement Algorithm 3, which interleaves three semi-decision procedures: Checking properties (P1) and (P2) for positive certificates of unboundedness, as well as looking for certificates of boundedness by looking for bounded regular over-approximations. Here we can check boundedness for the latter by Theorem 4. Note that while checking for (P1), it is possible to compute each language $L_a|_n$ explicitly since these languages are all finite. This is because, given any finite language $F \in C$ and an explicitly given finite language $A$, we know $F = A$ iff $F \cap (\Sigma_D^* \setminus A) = \emptyset$ and for all $w \in A$, $F \cap \{w\} = \emptyset$, where the first condition checks if $F \subseteq A$ and the second if $A \subseteq F$. Therefore, by enumerating all strings $w$, we can build $A$ iteratively.
Algorithm 3 Checking CB Boundedness

Input: Concurrent program \( \mathcal{P} = (D, \Sigma, (L_a)_{a \in \Sigma}, d_0, m_0) \) over \( C \) and context bound \( k \in \mathbb{N} \) run concurrently

begin
  /* (P1): Check if finite under-approximation is unbounded */
  foreach \( n \in \mathbb{N} \) do
    /* Explicitly find strings in \( L'_a|n \) */
    foreach \( a \in \Sigma \) do
      \( X_a \leftarrow \emptyset \), \( L'_a|n \leftarrow L_a \cap \Sigma_n \cap R_k \)
      foreach \( w \in \Sigma^* \) do
        if \( L'_a|n \cap \{w\} \neq \emptyset \) then
          \( X_a \leftarrow X_a \cup \{w\} \)
        if \( L'_a|n \cap (\Sigma_n^* \setminus X_a) = \emptyset \) then
          break
      end
    if \( \mathcal{P}_n = (D, \Sigma, (X_a)_{a \in \Sigma}, d_0, m_0) \) is unbounded then
      return \( \mathcal{P} \) unbounded.
  end

begin
  /* (P2): Check if unbounded segment can be reached */
  foreach prerun \( \rho \) of \( \mathcal{P} \), \( \langle \alpha \rangle \in \Sigma \), \( w \in aD(\Sigma \times \Sigma^*)^*(D \times D) \cup \{a\} \) do
    if \( \rho \) is a \( k \)-run that reaches \( c \) with \( w \in c \), \( w' = (d', d) \) where \( d' = c.d \), and \( \text{pref}(w^{|L_a|}) \cap \Sigma^* \) is infinite then
      return \( \mathcal{P} \) unbounded.
    if \( \rho \) is a \( k \)-run that reaches \( c \) with \( w \in c \), \( w = a \) where \( d' = c.d \), and \( \text{pref}((wd')^{|L_a|}) \cap \Sigma^* \) is infinite then
      return \( \mathcal{P} \) unbounded.
  end

begin
  /* Find a bounded over-approximation */
  foreach tuple \( (A_a)_{a \in \Sigma} \) of regular languages \( A_a \subseteq (\Sigma_n^* \cap R_k) \) do
    if \( (L_a \cap R_k) \cap (\Sigma_n^* \setminus A_a) = \emptyset \) for each \( a \in \Sigma \) then
      if \( \mathcal{P}' = (D, \Sigma, (A_a)_{a \in \Sigma}, d_0, m_0) \) is bounded then
        return \( \mathcal{P}' \) bounded.
  end

A Remark on Complexity  Our procedures show decidability, but do not provide complexity results. For particular classes of languages, precise complexity bounds are known. For example, CB Safety, CB Termination, and CB Boundedness for concurrent programs over regular languages are all \( \text{EXPSPACE} \)-complete [28], and over context-free languages are \( 2\text{EXPSPACE} \)-complete [11, 14]. These bounds use explicit constructions of the downclosure. In particular, our results show decidability of the same problems for concurrent programs over higher-order recursion schemes. However, we do not get an explicit complexity bound. While there is an explicit construction of the downclosure of these languages [53, 30, 20], a precise complexity bound for the construction remains open.

5 Further Results

Other Decision Problems  While we focus on safety, termination, and boundedness, there are decidability results for other properties and other classes of systems. The fair termination problem is a variant of termination, where we require that the scheduler is fair. Intuitively, a scheduler is fair if it schedules each partially executed thread that is infinitely often ready to execute. Context-bounded fair termination is decidable (but non-elementary) for context-free concurrent programs [15]. The problem is equivalent to Petri net reachability already for asynchronous programs [28]. It is undecidable for indexed languages.
Context-Bounded Analysis of Concurrent Programs

Context-bounded analysis has also been studied for non-regular specifications. Lal et al. [38] showed decidability for context-bounded analysis for a subclass of weighted pushdown systems. Recently, Baumann et al. [13] studied the context-bounded refinement problem for non-regular specifications. In their setting, there is a fixed number of recursive (context-free) threads which also generate a language over a set of events. The specification is given by a Dyck language. They show that checking containment in the specification is coNP-complete, the same complexity as that of context-bounded safety verification, albeit requiring very different techniques. An analogous result was shown for the setting of asynchronous programs, but the complexity is EXPSPACE-complete [12].

Tools and Sequentialization A practical motivation for studying context-bounded reachability was that, empirically, many bugs in concurrent programs could be found with a small number of context switches. This led to the development of several academic and industrial tools, such as CHESS [44] and CSeq [27]. CHESS incorporated context bounding in an enumerative search. CSeq and several other tools implemented sequentialization: a preprocessing step that compiles the original concurrent program into a sequential program that preserves all k-context bounded runs, an idea going back to Lal and Reps [37]. Context-bounding was integrated with other exploration heuristics such as abstract interpretation and partial-order reduction [45, 21, 41].

Context-Bounded Analysis of Related Models Context-bounding was studied for other models of concurrency, such as parameterized state machines communicating through message-passing over a given topology [18], concurrent queue systems [49], programs over weak memory models [9, 1], abstract models such as valence automata [43], etc. In each case, the notion of “context” has to be refined based on the model.

Similar Restrictions The theory of context-bounding has inspired other natural bounds in the analysis of concurrent systems. For example, a well-studied restriction is scope-bounding: In a k-scope-bounded run, there can be an unbounded number of context-switches, but during the time span of a single function call (i.e. between a push and its corresponding pop), there can be at most k interruptions [52]. This covers more executions than context-bounding, which comes at the cost of PSPACE-completeness of safety verification [52]. Scope-boundedness has also been studied in terms of timed systems [4, 17], temporal-logic model-checking [6], resulting formal languages [51], and as an under-approximation for infinite-state systems beyond multi-pushdown systems [48].

Similarly, a k-phase-bounded run consists of k phases, in each of which at most one stack is popped [50, 8]. Another variant is k-stage-bounded runs: They consist of k stages, each of which allows only one thread to write to the shared memory, whereas the other threads can only read from it [7]. Further restrictions are ordered multi-pushdown systems [19, 5] and delay-bounded scheduling [26].

Powerful abstract notions of under-approximate analysis (which explain decidability of several concrete restrictions described above) are available in the concepts of bounded tree-width [39] and bounded split-width [3, 23, 22].

In conclusion, context-bounding is an elegant idea that has been very influential both in practice and in theory. In practice, it has been incorporated in several tools for automatic analysis of programs. Theoretically, it has led to a wealth of new models and analysis algorithms. At this point, the theory has marched ahead of implementations: it is an interesting open challenge to see how far the new algorithms can also lead to practical tools.
References


3:14  

Context-Bounded Analysis of Concurrent Programs


Context-Bounded Analysis of Concurrent Programs


