Antichains for the Verification of Recursive Programs

Lukáš Holík[1] and Roland Meyer[2]

[1]Brno University of Technology [2]University of Kaiserslautern

Abstract. Safety verification of while programs is often phrased in terms of inclusions $L(A) \subseteq L(B)$ among regular languages. Antichain-based algorithms have been developed as an efficient method to check such inclusions. In this paper, we generalize the idea of antichain-based verification to verifying safety properties of recursive programs. To be precise, we give an antichain-based algorithm for checking inclusions of the form $L(G) \subseteq L(B)$, where G is a context-free grammar and B is a finite automaton. The idea is to phrase the inclusion as a data flow analysis problem over a relational domain. In a second step, we generalize the approach towards bounded context switching.

1 Introduction

We reconsider a standard task in algorithmic verification: model checking safety properties of recursive programs. To explain our model, assume we are given a recursive program P operating on a finite set of Boolean variables V. For this program, we are asked to check a safety property given by a finite automaton B. The task amounts to checking the inclusion

$$L(G(P)) \ \cap \ \bigcap_{v \in V} L(B(v)) \ \subseteq \ L(B).$$

Here, $G(P)$ is a context-free grammar whose language is the set of valid paths in the recursive program P. A path is valid if procedures are called and return in a well-nested manner. The context-free grammar only models the flow of control. It does not ensure the correct use of the Boolean variables. Indeed, there could be words in $L(G(P))$ where a write of 1 to variable v is followed by a read of value 0. To take data into account, we intersect the context-free language with a regular language $L(B(v))$ for each variable. The intersection only keeps words from $L(G(P))$ that obey the semantics of operations on the data domain. Such a separation of a program's semantics into the control and the data aspect is standard in verification [16].

The only non-regular part in the above inclusion is the control-flow language $L(G)$. We move the intersection with the regular languages to the right-hand side of the inclusion and obtain an equivalent formulation:

$$L(G(P)) \ \subseteq \ L(B) \ \cup \ \bigcup_{v \in V} \overline{L(B(v))}.$$

Since regular languages are closed under complement and closed under finite union, the right-hand side of the new inclusion is again a regular language $L(A)$.

The discussion allows us to define the safety verification problem for recursive programs that is the subject of this paper as follows. Given a context-free grammar G and a finite automaton A, check the inclusion

$$L(G) \;\subseteq\; L(A).$$

The classical algorithm for checking the inclusion determinizes A with the powerset construction, forms the complement, and computes the product with G. The result is the (context-free) emptiness problem $L(G \times \mathit{compl}(\mathit{det}(A))) = \emptyset$. The main bottleneck of this algorithm is the determinization of A. It may cause an exponential blow-up even in space, and makes the approach impractical. Actually, deciding the inclusion is PSPACE-complete.

For the simpler problem with finite automata on both sides, $L(A_1) \subseteq L(A_2)$, an efficient heuristics has been found. It is based on the so-called antichain principle [7,25] and prevents the state explosion in many practical cases. The observation is that the states in $A_1 \times \mathit{compl}(\mathit{det}(A_2))$ can be equipped with an ordering. For the emptiness check, it is sufficient to explore transitions from states that are minimal according to this ordering.

Our contribution is a generalization of the antichain principle to the problem $L(G) \subseteq L(A)$. The proposed algorithm computes a finite partitioning of Σ^* such that each partition contains words that are all accepted or all rejected by A. To test the inclusion, it is enough to test that G does not accept a word whose partition is rejected by A. The partitions are represented by sets of relations over states of the automaton A. Every relation encodes one possibility of how a word generated by a certain non-terminal of the grammar can influence the state of the automaton. This is the same concept as the one used in the proof that Büchi automata are closed under complement [5].

We formulate our algorithm via a reduction of $L(G) \subseteq L(A)$ to a data flow analysis problem $\mathit{DFA}(G, A)$. Implementing the antichain principle then amounts to modifying chaotic iteration so that it computes on a lattice of antichains [25] rather than a full powerset lattice. The reduction is theoretically appealing and allows for an elegant formulation of the antichain principle. Moreover, it reveals a close connection between automata theory and data flow analysis that opens up a possibility of using antichain optimizations in data flow analysis.

As a last step, we add parallelism to the picture. For multi-threaded recursive programs, safety verification can be formulated as

$$L(G_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup\, L(G_m) \;\subseteq\; L(A).$$

The problem is known to be undecidable [21]. For bug hunting, however, under-approximations have proven useful. The most prominent under-approximation is bounded context switching [20]. The observation made in practice is that bugs show up within few interactions among threads. If the threads share the same processor, this means bugs show up after few context switches — hence the name. Recall that a context switch occurs if one thread leaves the processor in order for another thread to be scheduled.

We propose a compositional approach to solving the above context-bounded inclusion. In a first step, we solve m data flow analysis problems $\mathit{DFA}(G_i, A)$, one for each grammar. In a second step, we combine the analysis results. The reduction $\mathit{DFA}(G_i, A)$ generalizes $\mathit{DFA}(G, A)$. We move from (sets of) relations to an analysis on (sets of) words of relations.

To sum up, the contributions of this paper are threefold:

1. The formulation of $L(G) \subseteq L(A)$ as a data flow analysis problem $\mathit{DFA}(G, A)$.
2. The antichain improvement of chaotic iteration for solving $\mathit{DFA}(G, A)$.
3. The generalization to bounded context switching.

Related work Antichain algorithms were first proposed in [7] in the context of solving games with imperfect information. The antichain principle was subsequently used to optimize language inclusion and universality checking of finite automata [25]. The basic idea of antichain algorithms, subsumption, is older and was used e.g. already in solving reachability of well-structured systems [12,1]. The antichain algorithms of [25] were further extended and improved in many ways. In our context, the generalization to the Ramsey-based universality and language inclusion checking for Büchi automata [13] is central. It introduces an ordering on what we call relations (in the related literature, the notion is called e.g. (super)graph, transition profile, or box). The problem of deciding language inclusion of nested word automata is closely related to the inclusion of a context-free language in a regular one, and is also similarly motivated. An extension of the Ramsey-based algorithm for nested word automata has been published in [14], and an alternative algorithm based on different principles, but using antichains, appears in [4].

Verification algorithms that use inclusion checking as a central subroutine have been proposed in [9,10,11]. These works focus on multi-threaded programs without recursion and develop complete algorithms (for inclusion). We tackle recursive programs, instead. In the multi-threaded setting, inclusion checking is undecidable [21] so that we consider an under-approximation of the problem. We show how to generalize the antichain principle to bounded context switching [20]. The approximation of bounded context switching has also been applied to relaxed memory models [2]. In the present work, however, we limit ourselves to Sequential Consistency.

In verification, the relational domain that we make use of is generalized to procedure summaries [24,22]. Summaries characterize the input-output relation of a procedure. They are not limited to tracking the state changes of a finite automaton. The language-theoretic view to verification problems is shared with [16,15,9,10,11,18,3]. We are not aware of any use of antichains for data flow analysis or of a formulation of regular inclusion as a data flow analysis problem. The domain for bounded context switching seems to be new. Our compositional analysis is related to the eager approach in [17].

2 Preliminaries

We introduce the three technical concepts used in this paper: regular inclusion, antichain algorithms, and data flow analysis.

2.1 CFG-REG

Given a finite alphabet Σ, we use Σ^* for the set of all finite words over Σ and write ε for the empty word. The concatenation of words $u, v \in \Sigma^*$ yields the word $u \cdot v \in \Sigma^*$. The shuffle of u and v is the set of interleavings of both words, for example $ab \sqcup\!\sqcup\, c = \{cab, acb, abc\} \subseteq \Sigma^*$.

A *context-free grammar* is a tuple $G = (\Sigma, X, x_0, R)$ where Σ is a finite alphabet and X is a finite set of non-terminals with initial symbol $x_0 \in X$. Component $R \subseteq X \times (\Sigma \uplus X)^*$ is the set of production rules. The language of G is defined using the derivation relation $\Rightarrow_G \subseteq (\Sigma \uplus X)^* \times (\Sigma \uplus X)^*$. Consider two words $\alpha, \beta \in (\Sigma \uplus X)^*$. Then $\alpha \Rightarrow_G \beta$ if there are $\alpha_1, \alpha_2 \in (\Sigma \uplus X)^*$, $x \in X$, and $(x, \gamma) \in R$ so that $\alpha = \alpha_1 \cdot x \cdot \alpha_2$ and $\beta = \alpha_1 \cdot \gamma \cdot \alpha_2$. We write $\Rightarrow_G^*$ for the reflexive and transitive closure of $\Rightarrow_G$. Moreover, if G is clear from the context we drop the subscript from $\Rightarrow_G$. The language of a non-terminal $x \in X$ is the set of words derivable from it:

$$L(x) \ := \ \{\alpha \in \Sigma^* \mid x \Rightarrow^* \alpha\}.$$

The *language of* G is $L(G) := L(x_0)$.

A context-free grammar is called *right-linear* if $R \subseteq X \times (\{\varepsilon\} \uplus (\Sigma \cdot X))$. Right-linear grammars correspond to finite automata where the non-terminals are the states, x_0 is the initial state, and the $x \in X$ with $(x, \varepsilon) \in R$ are the accepting states. This correspondence allows us to use the terminology for finite automata when talking about right-linear grammars. We shall use A, A_1, A_2 for right-linear grammars and G, G_1, G_2 for context-free grammars that are not necessarily right-linear. Throughout the paper, we use $G = (\Sigma, X, x_0, R)$ and $A = (\Sigma, Y, y_0, \rightarrow)$. Moreover, we write $y_1 \xrightarrow{a} y_2$ for $(y_1, a, y_2) \in \rightarrow$. We extend the notation to words: $y_1 \xrightarrow{w} y_2$ means there is a w-labeled path from y_1 to y_2.

Our contribution is an efficient algorithm for solving the following problem that we refer to as CFG-REG:

Given: A context-free grammar G and a right-linear grammar A.
Problem: Does $L(G) \subseteq L(A)$ hold?

As a running example, we consider $L(G_{ex}) \subseteq L(A_{ex})$ where the context-free grammar G_{ex} and the finite automaton A_{ex} are defined as follows:

$$\begin{array}{llllll}
G_{ex}: & x_0 \rightarrow a \cdot x_1 & x_0 \rightarrow b \cdot x_1 & A_{ex}: & y_0 \rightarrow a \cdot y_1 & y_0 \rightarrow b \cdot y_2 \\
 & x_1 \rightarrow c \cdot x_2 & x_1 \rightarrow d \cdot x_2 & & y_1 \rightarrow c \cdot y_3 & y_2 \rightarrow d \cdot y_4 \\
 & x_2 \rightarrow \varepsilon & & & y_3 \rightarrow \varepsilon & y_4 \rightarrow \varepsilon.
\end{array}$$

We have $L(G_{ex}) = \{ac, ad, bc, bd\}$ and $L(A_{ex}) = \{ac, bd\}$, so the inclusion fails. An attentive reader may notice that the context-free grammar G_{ex} is right-linear. This is only for the sake of simplicity, the algorithm of course works for any CFG, but G_{ex} allows to illustrate the main concepts well.

2.2 Antichain Algorithms

To explain the idea behind antichain algorithms, consider a simplified variant of CFG-REG where we check $L(A_1) \subseteq L(A_2)$ for given finite automata A_1 and A_2. The standard approach is to reformulate the inclusion as

$$L(A_1) \cap \overline{L(A_2)} = \emptyset.$$

Checking this emptiness involves determinizing A_2 using the powerset construction, which results in $det(A_2)$, complementing $det(A_2)$ by inverting the final states, giving $compl(det(A_2))$, and computing the product with A_1,

$$A_1 \times compl(det(A_2)).$$

The resulting automaton $A_1 \times compl(det(A_2))$ is then checked for emptiness. Unfortunate in this construction is that $det(A_2)$ may be exponential and that we need another product with the states of A_1.

Antichain algorithms check emptiness of $A_1 \times compl(det(A_2))$ on-the-fly. To explain the concept, we elaborate on the behavior of the product automaton. Let $A_i = (\Sigma, Y_i, y_{0,i}, \rightarrow)$ for $i = 1, 2$. States in the product automaton take the shape (y_1, Z_1) where $y_1 \in Y_1$ is a single state of A_1 and $Z_1 \subseteq Y_2$ is a set of states of A_2. We call (y_1, Z_1) a product state and Z_1 a macro state. Transitions in the product state (y_1, Z_1) are derived from transitions in y_1, because the macro state Z_1 in the determinized automaton can react to any input. If we have a transition $y_1 \xrightarrow{a} y_2$, then the product state takes a transition

$$(y_1, Z_1) \xrightarrow{a} (y_2, Z_2).$$

Here, Z_2 is the set of all states $z_2 \in Y_2$ that are reachable from some $z_1 \in Z_1$ with an a-labelled transition, $z_1 \xrightarrow{a} z_2$.

The goal is to disprove emptiness of $A_1 \times compl(det(A_2))$. This amounts to finding a product state (y, Z) where y is accepting in A_1 and Z is rejecting in the sense that it does not contain a final state of A_2. The larger the set Z becomes, the harder it is to avoid the final states of A_2. To disprove emptiness, we should thus only explore states that are minimal according to the following partial order $\leq$ on product states:

$$(y, Z) \leq (y', Z') \quad \text{if} \quad y = y' \text{ and } Z \subseteq Z'.$$

This is the idea of antichain algorithms: only explore states (y, Z) that are minimal according to $\leq$. The name stems from the fact that minimal elements in partial orders are incomparable, and sets of incomparable elements are also called antichains.

It remains to argue that we can safely discard larger states. We already discussed that state y_1 of A_1 determines the transitions of the product state (y_1, Z_1). Macro state Z_1 is guaranteed to have the successor state defined. A larger set Z'_1 will not enable or disable a transition. Moreover, the transition relation is monotone in the following sense. If $(y_1, Z_1) \xrightarrow{a} (y_2, Z_2)$ and we have $(y_1, Z_1) \leq (y_1, Z'_1)$, then $(y_1, Z'_1) \xrightarrow{a} (y_2, Z'_2)$ with $(y_2, Z_2) \leq (y_2, Z'_2)$. This means a larger state (y_1, Z'_1) again leads to a larger state (y_2, Z'_2) from which it is harder to accept than from (y_2, Z_2). In short, $(y, Z) \leq (y, Z')$ yields

$$L(y, Z') \subseteq L(y, Z)$$

for the product automaton $A_1 \times \mathit{compl}(\mathit{det}(A_2))$. So if we focus on minimal states, we are guaranteed to explore all behaviors.

2.3 Data Flow Analysis

Our presentation of data flow analysis follows standard textbooks [19,23]. A data flow analysis problem is given as a system of inequalities $\overline{\Delta} \geq Op(\overline{\Delta})$ that is interpreted over a domain of data flow values. The system of inequalities reflects the control flow in the program under scrutiny. For each location $l = 1, \ldots, n$ in the program there is a variable Δ_l. Intuitively, variable Δ_l records the analysis information obtained at location l. For each command c leading to l there is an inequality

$$\Delta_l \;\geq\; op_c(\Delta_1, \ldots, \Delta_n).$$

Operation op_c captures the effect that the command has on the already gathered analysis information. The inequality states that this effect should contribute to the analysis information at location l.

The domain that the system is interpreted over defines the actual analysis information being computed. As is common, we assume the domain to be a complete lattice $(D, \leq)$. A complete lattice is a partially ordered set where every subset of elements $D' \subseteq D$ has a least upper bound that we denote by $\sqcup D'$. As a second condition, the operations op_c used in the system of inequalities should be monotone, which means for all $d, d' \in D$ with $d \leq d'$ we have $op_c(d) \leq op_c(d')$.

A solution to the data flow analysis problem is a function *sol* that assigns to each variable Δ_l a value $sol(\Delta_l) \in D$ so that the inequalities are satisfied. We are interested in the least solution *lsol* wrt. a component-wise comparison of elements according to $\leq$. A unique least solution is guaranteed to exist by Knaster and Tarski's theorem.

If D is a finite set, the least solution to the system of inequalities can be computed with a Kleene iteration on D^n. This iteration, however, requires us to recompute, in every step, the analysis information for all n program locations — even if the analysis information has not changed. More efficient is the following algorithm, known as *chaotic iteration.* It is the algorithm we improve upon in this article:

```
$sol(\Delta_1) := \bot; \dots sol(\Delta_n) := \bot;$
**while** $\exists$ inequality with $sol(\Delta_l) \not\geq op_c(sol(\Delta_1), \dots, sol(\Delta_n))$ **do**
    $sol(\Delta_l) := sol(\Delta_l) \sqcup op_c(sol(\Delta_1), \dots, sol(\Delta_n));$
```

We start with all variables set to the least element in the lattice, denoted by $\bot$. As long as there is a command that can contribute to the value of a variable, we form the join to add the value.

Lemma 1 ([6]). *Consider $\overline{\Delta} \geq Op(\overline{\Delta})$ over a complete lattice $(D, \leq)$ where D is finite. Chaotic iteration terminates and gives the least solution lsol.*

3 From CFG-REG to Data Flow Analysis

We give a reduction of CFG-REG to a data flow analysis problem. It maps instance $L(G) \subseteq L(A)$ to the instance $DFA(G, A)$. Key to the reduction is an appropriate domain of data flow values. The idea is to determine the state changes that a word $w \in L(x)$ derived from a non-terminal $x \in X$ may induce on A. Phrased differently, the analysis considers words equivalent that induce the same state changes. In our running example $L(G_{ex}) \subseteq L(A_{ex})$, rule (x_2, ε) rewrites non-terminal x_2 to the empty word. As data flow information about x_2, we therefore add the relation $\rho(\varepsilon) = id$. Relation $\rho(\varepsilon)$ is indeed the identity as the empty word does not incur a state change. For a letter a, relation $\rho(a)$ contains precisely the pairs of states (y, y') so that y does an a-labeled transition to y'. In the running example, we have $\rho(a) = \{(y_0, y_1)\}$.

For the definition of $DFA(G, A)$, we formalize the mapping from words to relations over states as

$$\begin{aligned} \rho :\ & \Sigma^* \to \mathbb{P}(Y \times Y) \\ & w \mapsto \{(y_1, y_2) \mid y_1 \xrightarrow{w} y_2\}. \end{aligned}$$

Words are equipped with the operation of concatenation. Function ρ behaves homomorphically if we endow $\mathbb{P}(Y \times Y)$ with relational composition as operation. Recall that the relational composition of $\rho_1, \rho_2 \in \mathbb{P}(Y \times Y)$ is defined by

$$\rho_1; \rho_2 \ := \ \{(y_1, y_2) \mid \exists y : (y_1, y) \in \rho_1 \text{ and } (y, y_2) \in \rho_2\}.$$

With a component-wise definition, the operation carries over to sets of relations. The following lemma states that ρ is a homomorphism.

Lemma 2. *For all $w_1, w_2 \in \Sigma^*$, we have $\rho(w_1 \cdot w_2) \ = \ \rho(w_1); \rho(w_2)$.*

With this result, we only have to specify the data flow information for single letters. Relational composition will give us the analysis information for words.

The domain of data flow values is the complete lattice

$$(\mathbb{P}(\mathbb{P}(Y \times Y)), \subseteq).$$

The powerset $\mathbb{P}(\mathbb{P}(Y \times Y))$ contains all sets of relations between states in the given automaton. The domain is a standard powerset lattice with inclusion as ordering. It is complete. We operate on sets of relations to distinguish words that do not induce the same state changes. Consider the rules $(x_0, a \cdot x_1)$ and $(x_0, b \cdot x_1)$ in our running example $L(G_{ex}) \subseteq L(A_{ex})$. In the automaton, we have $(y_0, a \cdot y_1)$ and $(y_0, b \cdot y_2)$. A derivation of letter a induces the single state change $\rho(a) = \{(y_0, y_1)\}$ and similarly $\rho(b) = \{(y_0, y_2)\}$. There is, however, no word that admits a transition from y_0 to y_1 and from y_0 to y_2. Therefore, we cannot form the union of the relations $\rho(a)$ and $\rho(b)$ but have to compute on sets of relations $\{\rho(a), \rho(b)\}$. Indeed, for the running example an analysis based on relations rather than sets of relations gives incorrect results.

In data flow analysis, the current information is modified by operations *op*. In our setting, *op* forms a relational composition. With this in mind, the system of inequalities $DFA(G, A)$ for $L(G) \subseteq L(A)$ is defined as follows. We associate with every non-terminal $x \in X$ a variable Δ_x. Moreover, we use Δ_a for the set that only contains $\rho(a)$. Similarly, we let $\Delta_\varepsilon := \{id\}$. The system of inequalities contains one inequality for every rule in the grammar as follows. Let $(x, w) \in R$ with $w = w_1 \dots w_n \in (\Sigma \cup X)^*$. Then we have

$$\Delta_x \supseteq \Delta_{w_1}; \dots; \Delta_{w_n}.$$

The data flow analysis problem $DFA(G_{ex}, A_{ex})$ for our running example is

$$\begin{array}{ll}
\Delta_{x_0} \supseteq \{\{(y_0, y_1)\}\}; \Delta_{x_1} & \qquad \Delta_{x_0} \supseteq \{\{(y_0, y_2)\}\}; \Delta_{x_1} \\
\Delta_{x_1} \supseteq \{\{(y_1, y_3)\}\}; \Delta_{x_2} & \qquad \Delta_{x_1} \supseteq \{\{(y_2, y_4)\}\}; \Delta_{x_2} \\
\Delta_{x_2} \supseteq \{id\}. &
\end{array}$$

The least solution *lsol* to $DFA(G, A)$ has a well-defined meaning. It assigns to Δ_x precisely the relations $\rho(w)$ induced by the words w derivable from x.

Lemma 3. $lsol(\Delta_x) = \rho(L(x))$.

Proof. If $x = \varepsilon$ or $x = a \in \Sigma$ then the lemma trivially holds by the definition of Δ_x. Now consider $x \in X$. Note that x is not qualified further which means we reason simultaneously for all $x \in X$.

$lsol(\Delta_x) \subseteq \rho(L(x))$: Equivalently, for all $\rho \in lsol(\Delta_x)$, there is $w \in L(x)$ with $\rho(w) = \rho$. Assume that relation ρ is added to Δ_x within the k-th step of the Kleene iteration. We will proceed by induction on k. If $k = 0$, then the claim holds trivially since Δ_x is initialised as the empty set. Let the claim hold for $k \geq 0$. To show that it holds for $k + 1$, assume that ρ was added to Δ_x with the $(k + 1)$-st step of the Kleene iteration. It was constructed as $\rho = \rho_1; \dots; \rho_n$ due to an equation $\Delta_x \supseteq \Delta_{y_1}; \dots; \Delta_{y_n}$ where each ρ_i is either i. $\rho(\varepsilon)$ if $y_i = \varepsilon$, or ii. $\rho(a)$ if $y_i = a \in \Sigma$, or iii. is an element of $\rho(\Delta_{y_i})$ if $y_i \in X$. In the case iii., ρ_i was inserted into $\rho(\Delta_{y_i})$ within at most the k-th step of the Kleene iteration. By the induction hypothesis, this means that there is some $w_i \in L(y_i)$ with $\rho(w_i) = \rho_i$. This together with Lemma 2 gives that $\rho = \rho_1; \dots; \rho_n = \rho(w_1 \cdot \ldots \cdot w_n)$. By the definition of the system of equations, there is a rule $(x, y_1 \cdot \ldots \cdot y_n) \in R$ and hence $w_1 \cdot \ldots \cdot w_n \in L(x)$ by the definition of $L(x)$.

$\mathit{lsol}(\Delta_x) \supseteq \rho(L(x))$: Equivalently, for all $w \in L(x)$ we have $\rho(w) \in \mathit{lsol}(\Delta_x)$. We proceed by induction on the length ℓ of a derivation of w. For $\ell = 1$, the derivation uses only one rule, (x, ε) or (x, a) for some $a \in \Sigma$. Depending on the case, we have $\rho(w) = id$, or $\rho(w) = \rho(a)$. Due to the existence of the used rule, Δ_x will obtain $\rho(w)$ at the first step of the Kleene iteration. Assume the claim holds for the length of derivation $\ell > 1$. Let the first rule used be $(x, y_1 \cdots y_n)$ where $y_1, \ldots, y_n \in \Sigma \cup X$. Then w must be of the form $w_1 \cdot \ldots \cdot w_n$ where each $w_i \in \Sigma^*$ is i. the empty word, ii. a terminal, or iii. is obtained from y_i by a derivation of length at most ℓ. In case iii., $\rho(w_i) \in \mathit{lsol}(\Delta_{y_i})$ by the induction hypothesis. Because of this and the definitions of the system of inequalities and the composition of sets of relations, $\mathit{lsol}(\Delta_x)$ contains the composition $\rho(w_1); \ldots; \rho(w_n)$. By Lemma 2, $\rho(\Delta_x)$ contains $\rho(w_1 \cdot \ldots \cdot w_n) = \rho(w)$. □

A relation $\rho \subseteq Y \times Y$ is said to be *rejecting* if there is no pair $(y_0, y) \in \rho$ so that y is accepting. With Lemma 3, there is no accepting run of A on the words that induced the relation. Hence, the words belong to the complement of the automaton's language.

Theorem 1. *$L(G) \subseteq L(A)$ holds if and only if $\mathit{lsol}(\Delta_{x_0})$ does not contain a rejecting relation.*

4 Chaotic Iteration with Antichains

The previous section associates with each instance $L(G) \subseteq L(A)$ of CFG-REG a data flow analysis problem $\mathit{DFA}(G, A)$. Chaotic iteration as presented in the preliminaries computes a solution to such an analysis problem. We now improve upon chaotic iteration using the antichain principle.

When solving $\mathit{DFA}(G, A)$, chaotic iteration computes on sets of relations. Antichain algorithms compute on sets of incomparable elements. Together, this means we should replace the powerset domain $(\mathbb{P}(\mathbb{P}(Y \times Y)), \subseteq)$ used in our data flow analysis by a reduced domain of incomparable relations.

We construct the reduced domain in two steps. First, we note that relations $\rho_1, \rho_2 \in \mathbb{P}(Y \times Y)$ are partially ordered wrt. inclusion $\rho_1 \subseteq \rho_2$. The goal of chaotic iteration is to find a rejecting relation. To this end, it will be beneficial to focus on $\subseteq$-minimal elements. We therefore define the reduced domain to consist of all antichains of relations — sets of relations that are pairwise $\subseteq$-incomparable:

$$\mathbb{A}(\mathbb{P}(Y \times Y)) := \{\Delta \subseteq \mathbb{P}(Y \times Y) \mid \forall \rho_1, \rho_2 \in \Delta : \rho_1 \not\subseteq \rho_2\}.$$

In a second step, we lift the partial ordering $\subseteq$ on the set of relations to a partial ordering $\preceq$ on the set of antichains $\mathbb{A}(\mathbb{P}(Y \times Y))$ as follows:

$$\Delta_1 \preceq \Delta_2, \quad \text{if} \quad \forall \rho_1 \in \Delta_1 \; \exists \rho_2 \in \Delta_2 : \rho_1 \supseteq \rho_2.$$

For every relation $\rho_1 \in \Delta_1$ there is a $\subseteq$-smaller relation $\rho_2 \in \Delta_2$. Intuitively, this means Δ_2 is more likely to lead to a rejecting relation. We refer to the resulting partially ordered set as *antichain lattice*, similar to [25]. The following lemma justifies the name.

Lemma 4. $(\mathbb{A}(\mathbb{P}(Y \times Y)), \preceq)$ *is a complete lattice.*

Since the underlying set is finite, it is sufficient to show that joins exist. Let $\Delta_1, \Delta_2 \in \mathbb{A}(\mathbb{P}(Y \times Y))$. The least upper bound is

$$\Delta_1 \sqcup \Delta_2 := min(\Delta_1 \cup \Delta_2).$$

The main result states that chaotic iteration remains complete when we use the antichain lattice.

Theorem 2. *Chaotic iteration on DFA(G, A) is sound and complete when using the antichain lattice.*

When we restrict chaotic iteration to antichains, we consider a smaller domain of sets of relations. If we find a rejecting relation in this smalller domain, we find the rejecting relation in the larger domain. In this sense, our analysis is sound.

It remains to show completeness. If there is no rejecting relation with a chaotic iteration on antichains, then there is none when we iterate on the full powerset lattice. The following lemma is key to proving completeness. It states that $\subseteq$-minimal relations are sufficient for proving rejection and that inclusion is compatible with composition.

Lemma 5. *Consider $\rho_1 \subseteq \rho_2$. (1) If ρ_2 is rejecting, then ρ_1 is rejecting (2) For all $\rho \in \mathbb{P}(Y \times Y)$ we have $\rho_1; \rho \subseteq \rho_2; \rho$.*

To give an idea of why completeness holds, consider an arbitrary set of relations $\Delta \subseteq \mathbb{P}(Y \times Y)$ that may contain comparable elements. To obtain an antichain, we prune the set to the $\subseteq$-minimal relations. If there are rejecting relations in Δ, by Lemma 5(1) there is a minimal one. Moreover, with Lemma 5(2) continuing chaotic iteration on the minimal elements does not impair completeness.

5 Bounded Context Switching

We generalize our verification approach to multi-threaded recursive programs. In this setting, safety verification problems can be formulated as

$$L(G_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup L(G_m) \ \subseteq\ L(A).$$

Every context-free grammar $G_i = (\Sigma_i, X_i, x_{0,i}, R_i)$ represents (cf. Section 1) the valid control paths of a single thread in the multi-threaded program of interest. We can assume the threads to use different commands, which translates to disjointness of the alphabets, $\Sigma_i \cap \Sigma_j = \emptyset$ for all $i \neq j$. The shuffle operator $\sqcup\!\sqcup$ models the interleaving of computations from different threads. The task is to check whether every interleaving is valid wrt. the specification $A = (\Sigma, Y, y_0, \rightarrow)$ where $\Sigma = \biguplus \Sigma_i$. When modelling programs, specification A contains all feasible paths satisfying the verified property as well as all infeasible paths, where a read from a Boolean variable follows a write of the opposite value, like in Section 1.

The inclusion above, and hence safety verification of multi-threaded recursive programs, is undecidable [21]. A current trend in the verification community is to consider under-approximations of the problem that explore only a subset of the space of all computations. Under-approximations are good for bug hunting. If a bug is found in the restricted set of computations, it will remain present in the full semantics of the multi-threaded program. If the under-approximation is bug-free, however, we cannot conclude correctness of the program.

A prominent approach to under-approximation is *bounded context switching*. To explain the idea, assume the threads modeled by G_1 to G_m share the same processor. In this setting, a computation $w \in L(G_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup L(G_m)$ of the multi-threaded program may contain several context switches. (A context switch occurs if one thread leaves the processor and another thread is scheduled.) Phrased differently, the computation consists of several phases, $w = w_1 \cdot \ldots \cdot w_n$. In each phase, only one thread has the processor, without being preempted by another thread. Bounded context switching now only considers computations where each thread has at most k phases, for a given $k \in \mathbb{N}$. Note that the resulting set of computations is still infinite and even searches an infinite state space. What is limited is the number of interactions among threads. Indeed, the actions of one thread become visible to the other threads only at a context switch.

In the language-theoretic formulation, a context switch corresponds to an alphabet change. A phase is thus a maximal subword from a same alphabet. More formally, we say a word $w \in L(G_1) \sqcup\!\sqcup \ldots \sqcup\!\sqcup L(G_m)$ is *k-bounded*, $k \in \mathbb{N}$, if there are subwords $w = w_1 \cdot \ldots \cdot w_n$ so that

(1) for each w_i there is an alphabet $\Sigma_{\varphi(i)}$ with $w_i \in \Sigma^*_{\varphi(i)}$,
(2) $\Sigma_{\varphi(i)} \neq \Sigma_{\varphi(i+1)}$ for all $i \in [1, n-1]$,
(3) there are at most k subwords from each $\Sigma^*_{\varphi(i)}$.

We use $L_k(G_1, \ldots, G_m)$ to denote the set of all k-bounded words in the shuffle. The problem BCS-REG that is the subject of this section is defined as follows:

Given: Context-free grammars $G_1, \ldots, G_m$, a right-linear grammar A, and a number $k \in \mathbb{N}$.
Problem: Does $L_k(G_1, \ldots, G_m) \subseteq L(A)$ hold?

We propose a compositional approach to solving BCS-REG. We first approach the problem from the point of view of each single thread. Then we combine the obtained partial solutions into a solution for the overall multi-threaded program. A computation of the i-th thread is divided into k phases by context switches, which corresponds to a split of a word from $L(G_i)$ into k subwords. During each phase, the state of A is changed by the computation of the i-th thread. Between the phases, the environment, i.e., the other threads, change the state of A regardless of the i-th thread. We first, for every thread, compute a k-tuple of relations representing how the state of A can change during each of the k phases, assuming an arbitrary environment. The k-tuples will be obtained as a solution to the data flow analysis problem $\mathit{DFA}(G_i, A)$ in Section 3 that is now interpreted over a different domain.

To represent the behavior of the whole multi-threaded program, we shuffle the k-tuples of relations computed for the different threads. Such a shuffle represents an interleaving of the corresponding phases. From the point of view of a single thread, the shuffle concretizes a state change caused by *an arbitrary environment* to a state change caused by *a feasible computation of the other threads*. Finally, by composing the state changes caused by subsequent phases in the interleaving, we obtain a state change that is the overall effect of a computation of the whole multi-threaded program.

To better explain the idea and illustrate the technical development, consider the inclusion $L_2(G_1, G_2) \subseteq L(A)$ with

$$\begin{array}{llllllll} G_1: & x_{0,1} \to a_1 \cdot x_{1,1} & G_2: & x_{0,2} \to a_2 \cdot x_{1,2} & A: & y_0 \to a_1 \cdot y_1 & y_1 \to a_2 \cdot y_2 \\ & x_{1,1} \to b_1 & & x_{1,2} \to b_2 & & y_2 \to b_1 \cdot y_3 & y_3 \to b_2 \cdot y_4 \\ & & & & & & y_4 \to \varepsilon. \end{array}$$

Consider the word $a_1 \cdot a_2 \cdot b_1 \cdot b_2 \in L_2(G_1, G_2)$. It contains the phases a_1 and b_1 from $L(G_1)$. Phase a_1 induces the state change $\{(y_0, y_1)\}$ on the automaton A. Phase b_1 leads to $\{(y_2, y_3)\}$. Since we have to keep the state changes for each phase, the data flow analysis $\mathit{DFA}(G_1, A)$ determines (amongst others) the word of relations $\{(y_0, y_1)\} \cdot \{(y_2, y_3)\} \in \mathit{lsol}(\Delta_{x_0})$.

Technically, a *word of relations* is a word $\sigma = \rho_1 \cdot \ldots \cdot \rho_n$ whose letters are relations from $\mathbb{P}(Y \times Y)$. We use $\mathbb{P}(Y \times Y)^{\leq k}$ for the set of all such words of length up to $k \in \mathbb{N}$. We again construct a powerset lattice over this domain

$$(\mathbb{P}(\mathbb{P}(Y \times Y)^{\leq k}), \subseteq).$$

To re-use the system of inequalities from Section 3, we have to generalize the operation of relational composition to words of relations, $\sigma_1; \sigma_2$. The idea is to take a choice. Either we concatenate the words σ_1 and σ_2 or we compose the last relation in σ_1 with the first relation in σ_2. The former case reflects a context switch, the latter case occurs if there is no context switch (between the subwords inducing the last relation of σ_1 and the first relation of σ_2, respectively).

For the definition of this relational composition operator, consider the words of relations $\sigma_1 = \rho_{1,1} \cdot \ldots \cdot \rho_{1,k_1}$ and $\sigma_2 = \rho_{2,1} \cdot \ldots \cdot \rho_{2,k_2}$ in $\mathbb{P}(Y \times Y)^{\leq k}$. The operation of concatenation $\sigma_1 \cdot \sigma_2$ is defined as σ_1 and σ_2 are words. The composition of the last relation in σ_1 with the first relation in σ_2 is

$$\sigma_1 \circ \sigma_2 \ := \ \rho_{1,1} \cdot \ldots \cdot \rho_{1,k_1-1} \cdot (\rho_{1,k_1}; \rho_{2,1}) \cdot \rho_{2,2} \cdot \ldots \cdot \rho_{2,k_2}.$$

The relational composition $\sigma_1; \sigma_2$ yields the set of words (of relations) containing both $\sigma_1 \cdot \sigma_2$ and $\sigma_1 \circ \sigma_2$, provided the length constraint is met:

$$\sigma_1; \sigma_2 \ := \ \{\sigma_1 \cdot \sigma_2, \sigma_1 \circ \sigma_2\} \cap \mathbb{P}(Y \times Y)^{\leq k}.$$

With a component-wise definition, we lift the relational composition to sets of words of relations.

In the example, $\{\ \{(y_0, y_1)\}\ \}$ and $\{\ \{(y_2, y_3)\}\ \}$ are two sets each containing one word consisting of a single relation. Relational composition yields

$$\begin{aligned} lsol(\Delta_{x_{0,1}}) &= \{\ \{(y_0, y_1)\}\ \}; \{\ \{(y_2, y_3)\}\ \} \\ &= \{\ \{(y_0, y_1)\} \cdot \{(y_2, y_3)\}, \{(y_0, y_1)\} \circ \{(y_2, y_3)\}\ \} \\ &= \{\ \{(y_0, y_1)\} \cdot \{(y_2, y_3)\}, \emptyset\ \}. \end{aligned}$$

The empty set indicates that $a_1 \cdot b_1$ is not executable on the automaton A.

We use $DFA(G_i, A)$ for the data flow analysis problem that is derived from G_i and A using the above powerset lattice and the above relational composition. The following is the analogue of Lemma 3

Lemma 6. *$lsol(\Delta_x)$ contains precisely the words of relations induced by $L(x)$.*

Since words σ_1, σ_2 of relations are ordinary words over a special alphabet, also the shuffle operation $\sigma_1 ⧢ \sigma_2$ is defined. With reference to the above reduction,

$$lsol(\Delta_{x_{0,1}}) ⧢ \ldots ⧢ lsol(\Delta_{x_{0,m}})$$

yields precisely the words of relations that correspond to the k-bounded interleavings among words derivable in the different grammars.

For the desired inclusion $L_k(G_1, \ldots, G_m) \subseteq L(A)$, we check whether a word in the shuffle $lsol(\Delta_{x_{0,1}}) ⧢ \ldots ⧢ lsol(\Delta_{x_{0,m}})$ corresponds to a rejecting relation. To this end, we compose the relations in the word. The idea is that each word of relations in the shuffle corresponds to contiguous words in $L_k(G_1, \ldots, G_m)$. Therefore, we no longer have to deal with subwords. We define

$$eval(\rho_1 \cdot \ldots \cdot \rho_n) \ := \ \rho_1; \ldots; \rho_n.$$

Theorem 3. *Inclusion $L_k(G_1, \ldots, G_m) \subseteq L(A)$ holds if and only if there is no rejecting relation in $eval(lsol(\Delta_{x_{0,1}}) ⧢ \ldots ⧢ lsol(\Delta_{x_{0,m}}))$.*

To conclude the example, note that $lsol(\Delta_{x_{0,2}}) = \{\ \{(y_1, y_2)\} \cdot \{(y_3, y_4)\}, \emptyset\ \}$. Among other words, we have

$$\sigma := \{(y_0, y_1)\} \cdot \{(y_1, y_2)\} \cdot \{(y_2, y_3)\} \cdot \{(y_3, y_4)\} \in lsol(\Delta_{x_{0,1}}) ⧢ lsol(\Delta_{x_{0,2}}).$$

The evaluation yields $eval(\sigma) = \{(y_0, y_4)\}$. Still, the inclusion fails since we find the rejecting relation $eval(\emptyset \cdot \emptyset) = \emptyset \in eval(lsol(\Delta_{x_{0,1}}) ⧢ lsol(\Delta_{x_{0,2}}))$.

6 Conclusions and Future Work

We developed algorithms for the safety verification of recursive programs. This verification task is often phrased as an inclusion $L(G) \subseteq L(A)$ of a context-free language modeling the program of interest in a regular language representing the safety property. Our first contribution is a reformulation of the inclusion $L(G) \subseteq L(A)$ as a data flow analysis problem $DFA(G, A)$. The data flow analysis determines the state changes that the words derived in the grammar induce on the given automaton. This means the underlying domain of data flow values consists of sets of relations among states.

The data flow analysis problem *DFA*(G, A) can be solved by a standard algorithm called chaotic iteration. Our second contribution is an improvement of chaotic iteration. We show that the computation can be restricted to antichains of relations — while preserving completeness. Antichains are sets of relations that are pairwise incomparable. Phrased differently, our result reduces the powerset lattice used in *DFA*(G, A) to a lattice of antichains [25].

As a last contribution, we show how to generalize the appraoch to programs that are multi-threaded and recursive. While in this setting safety verification is known to be undecidable in general [21], an under-approximate variant of the problem remains decidable: Restricted to a bounded number of context switches, we can still check an inclusion $L(G_1) \sqcup \ldots \sqcup L(G_m) \subseteq L(A)$. Our approach is compositional in that it combines results from independent data flow analyses *DFA*(G_i, A). The reduction generalizes *DFA*(G, A) from (sets of) relations to (sets of) words of relations.

As an immediate task for future work, we will implement our antichain-based chaotic iteration and conduct an experimental evaluation. On the theoretical side, we plan to check whether a variant of the antichain principle can be used in related data flow analyses. One can also imagine importing algorithms to speed up the solution of *DFA*(G, A). An interesting candidate seems to be Newton iteration as presented in [8].

Acknowledgement. This work was supported by the Czech Science Foundation (projects 14-11384S and 202/13/37876P), the BUT FIT project FIT-S-14-2486, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

References

1. P.A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.
2. M. F. Atig, A. Bouajjani, and G. Parlato. Getting Rid of Store-Buffers in TSO Analysis. In *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.
3. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73. ACM, 2003.
4. Véronique Bruyère, Marc Ducobu, and Olivier Gauwin. Visibly pushdown automata: Universality and inclusion via antichains. In *LATA*, volume 7810 of *LNCS*, pages 190–201. Springer, 2013.
5. J.Richard Büchi. On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.
6. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Artificial Intelligence and Programming Languages*, pages 1–12. ACM, 1977.
7. Martin De Wulf, Laurent Doyen, and Jean-François Raskin. A lattice theory for solving games of imperfect information. In *HSCC*, volume 3927 of *LNCS*, pages 153–168. Springer, 2006.

8. J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *JACM*, 57(6), 2010.
9. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
10. A. Farzan, Z. Kincaid, and A. Podelski. Proofs that count. In *POPL*, pages 151–164. ACM, 2014.
11. A. Farzan, Z. Kincaid, and A. Podelski. Proof spaces for unbounded parallelism. In *POPL*, pages 407–420. ACM, 2015.
12. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2):63 – 92, 2001.
13. Seth Fogarty and MosheY. Vardi. Efficient büchi universality checking. In *TACAS*, volume 6015 of *LNCS*, pages 205–220. Springer, 2010.
14. Oliver Friedmann, Felix Klaedtke, and Martin Lange. Ramsey goes visibly push-down. In *ICALP*, volume 7966 of *LNCS*, pages 224–237. Springer, 2013.
15. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482. ACM, 2010.
16. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *CAV*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.
17. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, volume 4963 of *LNCS*, pages 282–298. Springer, 2008.
18. Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In *FASE*, volume 7212 of *LNCS*, pages 362–376. Springer, 2012.
19. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
20. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
21. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Tr. on Prog. Lang. and Sys.*, 22(2):416–430, 2000.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM, 1995.
23. H. Seidl, R. Wilhelm, and S. Hack. *Compiler Design - Analysis and Transformation.* Springer, 2012.
24. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. Technical Report 2, New York University, 1978.
25. M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.