Foundations of Infinite-State Verification

Rupak MAJUMDAR
Max Planck Institute for Software Systems, Germany; Email: rupak@mpi-sws.org.

Abstract. These lecture notes give an introduction to the field of infinite-state model checking. We take a language-theoretic view, and focus on a few foundational results.

Keywords. Infinite-state model checking, verification.

1. Introduction

The initial applications and successes of model checking techniques were in verifying finite-state hardware circuits or finite-state descriptions of communication protocols. Since then, it has been applied to many different domains, such as software implementations, real-time and hybrid systems, and parameterized families of circuits and protocols, which are not finite state. For example, software implementations contain potentially unbounded data (counters, or heap data structures) and control (function stack, dynamically allocated threads of execution), real-time systems manage clocks, hybrid systems model interactions with continuous physical processes, and parameterized protocols define an infinite family of protocols, one for each setting of the parameters. Unlike the finite-state case, one cannot expect a generic decidable model checking algorithm for infinite-state systems: the reachability problem for Turing machines is undecidable, and many infinite state systems can simulate Turing machines.

The purpose of these lecture notes is to understand how model checking techniques can be extended to the infinite-state setting, to identify special cases where the model checking problem remains decidable, despite the infinite state space, and to describe useful heuristics that work well in practice, even if the underlying problem is undecidable. Throughout, I have tried to emphasize the language-theoretic connections that underlie the decidability (and approximation) results.

The material is structured as follows. Section 2 sets up the stage for infinite-state model checking, and gives a general condition for termination (bisimulation relations of finite index). Section 3 covers abstract model checking, arguably the most common approach to infinite-state model checking, as well as counterexample-guided abstraction refinement. Section 4 introduces well-structured transition systems, and gives a general decidability result for reachability analysis. Section 5 considers pushdown reachability, which underlies interprocedural analysis of programs. I take a language-theoretic view and connect the results to well-known results in formal language theory. Finally, Section 6 looks at model checking concurrent programs. I show how the different techniques from Sections 3-5 come together to prove a decidability result for the safety verification of asynchronous programs with Boolean data.

In these notes, I assume familiarity with basic model checking, e.g., at the level of Clarke, Grumberg, and Peled's *Model Checking*, or Baier and Katoen's *Principles of Model Checking*. I also assume familiarity with formal language theory, at the level of Sipser's *An Introduction to the Theory of Computation*. For brevity, "proofs" are really "proof sketches" (with pointers to references), and we omit some standard definitions that may be found in the above text books.

Acknowledgments. These lecture notes grew out of a series of lectures presented at the Marktoberdorf Summer School in Summer 2013. I thank the organizers for inviting me to the event and for the participants in the summer school for commenting on the material.

2. A Quick Recap: Transition Systems and Invariants

Let me start by recalling some basic terminology from model checking.

A transition system $\mathcal{S} = (S, S_0, A)$ consists of a (not necessarily finite) set S of states, a set $S_0 \subseteq S$ of initial states, and a set A of *transitions*. Each transition $a \in A$ is a binary relation on S. We write $s \xrightarrow{a} t$ if $(s,t) \in a$. When S is finite, we say $\mathcal{S}$ is a *finite* transition system. Given a state $s \in S$ and a transition $a \in A$, we call the set $\{t \mid s \xrightarrow{a} t\}$ the *a-successors* of s. For a set $T \subseteq S$ of states and transition $a \in A$, we define the two operations

$$\mathsf{post}(T, a) = \{s \mid \exists s' \in T . s' \xrightarrow{a} s\}$$

which describes the set of states reachable from a set T in one step by executing transition a, and

$$\mathsf{pre}(T, a) = \{s \mid \exists s' \in T . s \xrightarrow{a} s'\}$$

which describes the set of states that can reach T in one step by executing transition a.

A *run* of a transition system is a (possibly infinite) sequence $s_0, s_1, \ldots$ such that $s_0 \in S_0$, and for each $i \geq 0$, there is an $a_i \in A$ such that $s_i \xrightarrow{a_i} s_{i+1}$. A state $s \in S$ is *reachable* if there is a run $s_0, s_1 \ldots, s_k$ such that $s_k = s$. Let Reach denote the set of all reachable states.

For a transition system $\mathcal{S} = (S, S_0, A)$, a set $T \subseteq S$ of states is an *invariant* of $\mathcal{S}$ if every reachable state belongs to T, that is, if $\mathsf{Reach} \subseteq T$. The invariant verification problem asks, given a transition system $\mathcal{S}$ and a set of states T, whether T is an invariant of $\mathcal{S}$. Invariant verification is a fundamental problem in system verification, and it can be shown that any safety property of the system (intuitively, properties that assert that nothing bad happens) can be reduced to invariant verification.

Example: Programs and Control Flow Graphs We model sequential programs using control flow graph representations. A *program* $\mathsf{P} = (x, \mathsf{locs}, \ell_0, \mathcal{T})$ consists of a set x of variables, a set locs of *control locations*, an initial location $\ell_0 \in \mathsf{locs}$, and a set $\mathcal{T}$ of *transitions*. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ') where $\ell, \ell' \in \mathsf{locs}$ are control flow locations, and ρ is a constraint over free variables from $x \cup x'$, where the variables from x' denote the values of the variables from x in the next state.

Algorithm 1 Enumerative reachability algorithm

Input: transition system $\mathcal{S} = (S, S_0, A)$, set of states $T \subseteq S$
Output: "yes" if T is an invariant, "no" otherwise

set Reachable, **multiset** Frontier
Reachable $= \emptyset$;
Frontier $= S_0$
while Frontier $\neq \emptyset$ **do**
  choose s from Frontier; Frontier $=$ Frontier $\setminus \{s\}$
  if $s \notin T$ **then**
    return "no"
  end if
  if $s \notin$ Reachable **then**
    Reachable $=$ Reachable $\cup \{s\}$
    foreach $a \in A$, add all $t \in S$ such that $s \xrightarrow{a} t$ to Frontier
  end if
end while
return "yes"'

As concrete examples of the relation ρ, consider an imperative programming language with assignment operations $y := exp$ and conditionals $\texttt{assume}(bexp)$, for expressions exp and predicates $bexp$. The relation ρ for the assignment statement is

$$y' = exp \wedge \bigwedge_{z \in x, z \neq y} z' = z$$

and for the conditional statement is

$$bexp \wedge \bigwedge_{y \in x} y' = y$$

A program defines a transition system in the following way. A state of the program P consists of its location $\ell \in$ locs and a valuation of the variables from x. The set of initial states consist of the initial location ℓ_0 and an arbitrary valuation to the variables. Each edge $e = (\ell, \rho, \ell')$ in the control flow graph gives rise a transition $(\ell, v) \xrightarrow{e} (\ell', v')$ where $\rho(v, v')$ holds.

2.1. Enumerative Reachability

One way to check if T is invariant is to explicitly compute the set Reach of reachable states and check that each reachable state belongs to T. In case $\mathcal{S}$ is finite, one can compute Reach by a graph reachability algorithm. Figure 1 gives a simple description of such an algorithm.

The graph reachability algorithm maintains two data structures: a set data structure Reachable to store the set of states already found to be reachable, and a multiset data structure Frontier to store the states that need to be explored. Initially, Frontier contains all the initial states from S_0 and Reachable is empty. The reachability algorithm is a loop that runs while Frontier is not empty. The algorithm maintains the property that

every state in Reachable as well as every state in Frontier is reachable. In each iteration, the algorithm removes a state s from Frontier and checks if $s \in T$. If not, T cannot be an invariant. Otherwise, it checks if s is already known to be reachable (i.e., if $s \in$ Reachable). If so, it proceeds to the next iteration. Otherwise, if $s \notin$ Reachable, the algorithm adds s to Reachable and for all $a \in A$, adds all a-successors of s to Frontier. On termination, Reachable consists of the set of reachable states, and since the check $s \notin T$ never failed, all these states are known to be a subset of T. Different implementations of the data structures gives graph traversal strategies such as depth-first search (Frontier maintained as a stack) or breadth-first search (Frontier maintained as a queue).

The graph reachability algorithm runs in linear time in the size of the graph. Of course, if the transition system is not finite, then there is no guarantee that graph reachability will terminate (it terminates in the simple case when the set of reachable states is finite). Even when the transition system is finite, going over the states one at a time can be extremely time consuming. This motivates the use of *symbolic techniques* which look at *sets* of states at a time.

2.2. Symbolic Reachability

The core idea of *symbolic* techniques is to represent sets of states of a transition system using formulas in some logic, and performing operations on sets of states by manipulating logical formulas.

We define symbolic transition systems using a set of predicates and functions coming from a *state vocabulary* $\mathcal{S}$ and a fixed set $\mathcal{D}$ on which the vocabulary will be interpreted. A *state formula* is a formula over the state vocabulary. For a state vocabulary $\mathcal{S}$, we write $\mathcal{S}'$ for the vocabulary in which each symbol x from $\mathcal{S}$ is given a new name x'. We extend the priming notation to formulas: the $\mathcal{S}'$-formula φ' is obtained from the $\mathcal{S}$-formula φ by substitution each symbol $x \in \mathcal{S}$ by the primed version x'.

A *symbolic transition system* $\mathcal{S}_\star = (\langle \mathcal{S}, \mathcal{D} \rangle, \mathsf{Init}, \mathcal{A})$ consists of a state vocabulary $\mathcal{S}$ and a set $\mathcal{D}$, a state formula Init (called the *initial condition*), and a set $\mathcal{A}$ of *(symbolic) actions*. Each symbolic action $a \in \mathcal{A}$ is a formula over the vocabulary $\mathcal{S} \cup \mathcal{S}'$.

A symbolic transition system represents a transition system (S, S_0, A) in the following way. The set S of states consists of all interpretations to the vocabulary $\mathcal{S}$ in the set $\mathcal{D}$. A state is initial if it satisfies the initial condition Init. The set of actions A consists of the binary relations induced by the formulas a in $\mathcal{A}$. That is, for two states s, t and an action $a \in \mathcal{A}$, we have $s \xrightarrow{a} t$ if $\langle s, t' \rangle$ satisfies the $\mathcal{S} \cup \mathcal{S}'$-formula a, where t' is an interpretation of all symbols in $\mathcal{S}'$ obtained by assigning the value $t(x)$ to each symbol $x' \in \mathcal{S}'$ on a copy of $\mathcal{D}$. Under this interpretation, state formulas define sets of states of the underlying transition system; for a state formula ψ, we write $[\![\psi]\!]$ to denote this set of states.

We write $\varphi \models \psi$ to denote the set of states represented by φ is a subset of the set of states represented by ψ, and we use standard logical connectives. For a state formula φ and action $a \in \mathcal{A}$, we define $\mathsf{post}(\varphi, a)$ and $\mathsf{pre}(\varphi, a)$ as the state formulas

$$\mathsf{post}(\varphi, a) \equiv (\exists \mathcal{S}. \varphi \wedge a)[\mathcal{S}/\mathcal{S}']$$
$$\mathsf{pre}(\varphi, a) \equiv \exists \mathcal{S}'. a \wedge \varphi'$$

where the existential quantifer "$\exists \mathcal{S}$" quantifies out all symbols in $\mathcal{S}$, and the substitution "$[\mathcal{S}/\mathcal{S}']$" substitutes all symbols in $\mathcal{S}'$ with their unprimed version from $\mathcal{S}$. Seman-

tically, $[\![\mathsf{post}(\varphi, a)]\!] = \mathsf{Post}([\![\varphi]\!], a)$ and $[\![\mathsf{pre}(\varphi, a)]\!] = \mathsf{Pre}([\![\varphi]\!], a)$. We write $\varphi\langle a\rangle\psi$ as shorthand for $\mathsf{post}(\varphi, a) \models \psi$, or equivalently, $\varphi \wedge a \models \psi'$.

A state formula φ is an invariant if every reachable state of the underlying transition system satisfies φ. The *invariant verification problem* asks, given a symbolic transition system $\mathcal{S}_\star$ and a state formula ψ, if ψ is an invariant of $\mathcal{S}_\star$.

Given a symbolic transition system, one can compute the set of reachable states as follows, assuming decision procedures for the underlying logic. The set of states reachable in at most 0 steps, Reach_0, is just the set of initial states. So we define $\mathsf{Reach}_0 = \mathsf{Init}$. Having defined Reach_i, we define the set of states reachable in at most $i+1$ steps as $\mathsf{Reach}_{i+1} = \mathsf{Reach}_i \vee \bigvee_{a\in\mathcal{A}} \mathsf{post}(\mathsf{Reach}_i, a)$. If for some N, we find that $\mathsf{Reach}_{N+1} \models \mathsf{Reach}_N$, then we have reached a fixed point in the iteration, and $\mathsf{Reach} = \mathsf{Reach}_N$. For a state formula ψ, we have that ψ is an invariant iff $\mathsf{Reach} \models \psi$.

Reachability analysis defined using the post operator is called *forward* analysis, as it starts with the initial states and explores the transition graph "forward." An analogous *backward* analysis can be defined as follows. We define $\mathsf{Breach}_0 = \neg\psi$, the set of "bad" regions, and iterate $\mathsf{Breach}_{i+1} = \mathsf{Breach}_i \vee \bigvee_{a\in\mathcal{A}} \mathsf{pre}(\mathsf{Breach}_i, a)$. To ensure that ψ is an invariant, we check that the fixed point of the iteration does not intersect the initial states; i.e., ψ is an invariant iff $\cup_{i\in\mathbb{N}}\mathsf{Breach}_i \cap \mathsf{Init}$ is not satisfiable.

Instead of full reachability analysis, which checks if there is a path of some length, sometimes one is interested in checking if a path of length exactly $k \geq 0$ can reach a state not in ψ. This problem can be reduced to the satisfiability problem for the underlying logic in the following way. Let $T(\mathcal{S}, \mathcal{S}') = \bigvee\{a(\mathcal{S}, \mathcal{S}') \mid a \in \mathcal{A}\}$, where we have explicitly indicated the vocabulary $\mathcal{S} \cup \mathcal{S}'$ in the formula. We introduce the *priming notation*: the vocabulary $\mathcal{S}^{'(n)}$ is a disjoint copy of the vocabulary $\mathcal{S}$ with n primes attached to each non-logical symbol. We can construct the formula

$$\mathsf{Init}(\mathcal{S}) \wedge T(\mathcal{S}, \mathcal{S}') \wedge \ldots \wedge T(\mathcal{S}^{'(k-1)}, \mathcal{S}^{'(k)}) \wedge \neg\psi[\mathcal{S}^{'(k)}/\mathcal{S}] \tag{1}$$

and ask if it is satisfiable (over the vocabulary $\mathcal{S} \cup \ldots \mathcal{S}^{'(k)}$). The above formula unrolls the transition relation for k steps, and checks if the last state along the run is outside ψ. This idea of reducing the search for bad paths to satisfiability of formulas is called *bounded model checking*, and has been very successfully applied to bug finding in hardware and software [7].

It can happen that each Reach_i is representable in the logic, but their (infinite) union, the set of reachable states, is not. In that case, the iterations can continue forever. This is not surprising, since reachability analysis for most models is undecidable, even if the k-step reachability relation is decidable.

2.3. Examples of Symbolic Transition Systems

Boolean Systems and BDDs Let X be a set of Boolean variables. A transition system whose states consist in valuations to variables in X can be encoded symbolically using propositional formulas and using satisfiability procedures for propositional logic. The encoding of symbolic transition systems using propositional logic represented using binary decision diagrams (BDDs) [8] was a key step in the industrialization of model checking, and in fact, symbolic model checking was synonymous with model checking Boolean symbolic transition systems using BDDs [9,28]. For bounded model checking,

one can use propositional satisfiability checkers to check if the formula in Equation (1) is satisfiable.

Timed Automata Timed automata [2] are models of timed systems that incorporate discrete automata with real-valued clocks. At each location of the automaton, the clocks increase in value at a constant rate. Based on the values of the clocks, discrete edges can be taken to new locations, and on taking a discrete edge, some clocks can be reset. A symbolic representation for clock values of timed systems can be given using *difference constraints* of the form $x - y \sim c$, where x, y are real-valued variables, c is a constant, and $\sim \in \{\leq, <, =, >, \geq\}$. A set of states is then represented as a list of pairs, the first part of the pair is the discrete location and the second part is a conjunction of difference constraints over the clock variables.

2.4. Termination: Bisimulation

For finite-state systems, invariant verification is trivially decidable. In some infinite models, reachability remains decidable. One way to prove decidability is to build a quotient transition system based on an appropriate equivalence relation on states.

Let $\mathcal{S}$ be a transition system and T a set of states. Two states s and t are related by a *bisimulation* relation if the following conditions hold: (a) either both $s \in T$ and $t \in T$ or both $s \notin T$ and $t \notin T$, and either both $s \in S_0$ and $t \in S_0$ or neither, (b) for each s' such that $s \xrightarrow{a} s'$ there is a t' such that $t \xrightarrow{a} t'$ and s' and t' are related by a bisimulation relation, and (c) for each t' such that $t \xrightarrow{a} t'$ there is a s' such that $s \xrightarrow{a} s'$ and s' and t' are related by a bisimulation relation.

Exercise 1 *Bisimulation is an equivalence relation on states.*

Given an equivalence relation $\equiv$ on states, define the *quotient* transition system $\mathcal{S}_{\equiv}$ as follows. The states of the quotient are equivalence classes of $\equiv$. The initial states are the equivalence classes of states in S_0, i.e., $\{[s]_{\equiv} \mid s \in S_0\}$, where $[s]_{\equiv}$ denotes the equivalence class of s. The transition relation takes $X \xrightarrow{a} Y$ iff there is some state $s \in X$ and some state $t \in Y$ such that $s \xrightarrow{a} t$.

Theorem 1 *Let $\mathcal{S}$ be a transition system and $\equiv$ an equivalence relation on states. Let T be a set of states. Then T is reachable in $\mathcal{S}$ if $[T] = \{[t]_{\equiv} \mid t \in T\}$ is reachable in $\mathcal{S}_{\equiv}$. If $\equiv$ is a bisimulation relation, then T is reachable in $\mathcal{S}$ iff $[T] = \{[t]_{\equiv} \mid t \in T\}$ is reachable in $\mathcal{S}_{\equiv}$.*

Now, if $\equiv$ is a bisimulation relation with a finite number of equivalence classes, then the quotient is a finite-state system on which reachability can be performed. The reachability analysis would decide reachability on the original system. The existence of bisimulation relations of a finite index is used to prove that backward reachability analysis terminates for timed automata [2,22,21].

In fact, the existence of a bisimulation relation of finite index can be used to model check more expressive logics (see [21] for details).

3. Abstraction

3.1. Inductive Invariants and Abstract Reachability Graphs

In general, reachability does not terminate for programs (indeed, just two integer valued variables is enough for undecidability [30]). Hence, we turn to heuristics that, while incomplete in general, work well in practice.

First, notice that computing the set of reachable states is often overkill to solve the invariant verification problem. Instead of computing the exact set of reachable states, and checking that this set is contained in ψ, one can prove that ψ is an invariant by devising an *inductive invariant* ϕ and checking the following conditions:

(I1: Initiation) $\mathsf{Init} \models \phi$;
(I2: Inductiveness) for each $a \in A$, we have $\phi\langle a\rangle\phi$; and
(I3: Safety) $\phi \models \psi$.

By induction, it is easy to show that for any ϕ satisfying (I1) and (I2), we have that $\mathsf{Reach} \models \phi$. Together with condition (I3), this entails that ψ is an invariant.

Given a candidate inductive invariant ϕ, the checks (I1)-(I3) can be discharged using a decision procedure for the underlying logic. The *invariant synthesis* problem is to construct a suitable inductive invariant. We introduce abstract reachability graphs as a first step toward invariant synthesis.

Let $\mathcal{S}_\star = (\langle \mathcal{S}, \mathcal{D}\rangle, \mathsf{Init}, \mathcal{A})$ be a symbolic transition system and let ψ be a state formula. An *abstract reachability graph* (ARG) $G = \langle V, E, \mathsf{r}, \Phi\rangle$ is a rooted, directed, labeled graph with a set V of nodes, a set of *transition* edges $E \subseteq V \times \mathcal{A} \times V$, a special root node $\mathsf{r} \in V$, and a node-labeling function Φ mapping each node in V to a state formula.

We write $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ for $(\mathsf{n}, a, \mathsf{n}_1) \in E$. For a node $\mathsf{n} \in V$, we say n is *reachable from* r and write $\mathsf{r} \xrightarrow{\star} \mathsf{n}$ if there is some $\ell \geq 0$ and a path $\mathsf{r} \xrightarrow{a_1} \mathsf{n}_1 \xrightarrow{a_2} \ldots \xrightarrow{a_\ell} \mathsf{n}$ of E-edges from r to n.

An ARG is *well-labeled* if the following conditions hold:

WL_1 $\mathsf{Init} \models \Phi(\mathsf{r})$;
WL_2 For each edge $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ in E, $\Phi(\mathsf{n})\langle a\rangle\Phi(\mathsf{n}_1)$; and
WL_3 For each n such that $\mathsf{r} \xrightarrow{\star} \mathsf{n}$, we have $\Phi(\mathsf{n}) \models \psi$.

An ARG is *complete* if for each node $\mathsf{n} \in V$ such that $\mathsf{r} \xrightarrow{\star} \mathsf{n}$ and for each transition $a \in A$, there is a $\mathsf{n}_1 \in V$ such that $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ is in E.

Theorem 2 *Let $\mathcal{S}_\star$ be a symbolic transition system and ψ a state formula. If there exists a well-labeled and complete ARG for $\mathcal{S}_\star$ and ψ, then ψ is an invariant of $\mathcal{S}_\star$.*

PROOF. We claim that

$$\bigvee\{\Phi(\mathsf{n}) \mid \mathsf{n} \in V, \mathsf{r} \xrightarrow{\star} \mathsf{n}\}$$

satisfies conditions I_1, I_2, and I_3. Condition I_1 holds because $\mathsf{Init} \models \Phi(\mathsf{r})$ by the well-labeling condition. Condition I_2 holds because (by well-labeling) $\Phi(\mathsf{n})\langle a\rangle\Phi(\mathsf{n}_1)$ holds for each edge $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ for each n reachable from r, and moreover, by completeness, each

node reachable from the root node has an outgoing edge for each $a \in A$. Condition I_3 holds because $\Phi(\mathsf{n}) \models \psi$ for each n reachable from r by the well-labeling condition. $\square$

3.2. Abstraction

The key observation that makes the ARG useful is that for any edge $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$, the label $\Phi(\mathsf{n}_1)$ need not be *exactly* $\mathsf{post}(\Phi(\mathsf{n}), a)$, but should *contain* $\mathsf{post}(\Phi(\mathsf{n}), a)$. This opens up the possibility of *approximate*, or abstract, computations of reachable sets. As long as the abstractions do not lose too much precision, in the sense of condition WL_3, one can use the approximations to check if ψ is an invariant.

We now give an algorithm to construct ARGs through a non-deterministic algorithm, Algorithm AbstractSafety. Its inputs are a symbolic transition system $\mathcal{S}_\star$ and a state formula ψ. We assume that $\mathsf{Init} \models \psi$ (otherwise, we stop immediately and return that ψ is not an invariant).

Initially, the algorithm starts with an ARG with one node: the root node r labeled Init. Initially, there are no edges, i.e., $E = \emptyset$. This graph is well-labeled, but not complete if $A \neq \emptyset$.

In each step, it picks a node n in the ARG and a transition $a \in \mathcal{A}$ such that n has no outgoing edge labeled with a. It picks a state predicate ϕ such that $\Phi(\mathsf{n})\langle a\rangle\phi$ holds. If $\phi \wedge \neg\psi$ is satisfiable, the algorithm stops with an error.

Otherwise, if there is already a node n_1 in the ARG labeled ϕ, the algorithm adds the edge $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ to E. Otherwise, if there is no such node, it adds a new node n_2 to the ARG and labels it with ϕ.

Each step of the algorithm makes the graph "more complete." The algorithm terminates, and states ψ is an invariant, if there is no node n and action a such that node n has no outgoing a-edge in E.

Proposition 1 *[Soundness] If Algorithm* AbstractSafety*, on input $\mathcal{S}_\star$ and ψ, terminates and states ψ is an invariant, then ψ is an invariant of $\mathcal{S}_\star$.*

The proof of the proposition follows by checking the properties WL_1-WL_3 of the ARG at the end of the computation. When the algorithm returns an error, we can additionally produce a *possible counterexample*: a list of transitions labeling the path from the root node until the node at which error was raised, together with the transition for which the error was raised. Note that a possible counterexample need not be a "real" counterexample of the system: the process of choosing ϕs introduces approximations, and the counterexample produced by the algorithm may not be feasible in $\mathcal{S}_\star$.

We must answer two key steps to implement the algorithm: what strategy should we use to expand nodes, and how do we choose ϕ? The first question is technically not very deep (choose any graph traversal strategy), but can have practical consequences on the efficiency and scalability of the algorithm.

The theory of abstract interpretation [10] formalizes the second question in terms of fixing abstract domains and computing "best" approximations relative to the abstract domains. Instead of giving the general theory, we give two examples.

Polyhedral Abstraction In *polyhedral abstraction*, we look for state formulas defined by linear constraints over constants in the vocabulary, that is, polyhedral sets in the n-dimensional space of program variables. Using efficient algorithms for polyhedral manipulation, one can implement the logical operations effectively: the conjunction operation is polyhedral intersection, the disjunction operation either keeps an explicit list of polyhedra or, to make the algorithm efficient, takes the convex hull of the operands, and satisfiability checking determines if a polyhedron is non-empty. If the transition relations are defined by linear constraints, then one can compute the pre and post operations using intersections and projections of polyhedra.

Polyhedral abstractions have been successfully used to verify properties of programs, such as array bounds checks or error bounds in numerical computations. Notice that the ARG construction need not terminate when using a polyhedral abstraction. To ensure termination, a *widening* operation, that guarantees syntactically that increasing chains stabilize in a finite number of steps, is used.

Faster, but less expressive, abstract domains that can represent a subclass of polyhedra, such as intervals (of the form $c_1 \leq x \leq c_2$), difference constraints (of the form $x - y \leq c$), or octagons (of the form $x \pm y \leq c$) have been used as well.

Predicate Abstraction In *predicate abstraction*, we fix a finite set Π of first order formulas over the vocabulary $\mathcal{S}$ and consider of the lattice of Boolean formulas over Π ordered by implication. The predicate abstraction of a state formula ψ with respect to the set Π of predicates is the smallest (in the implication ordering) state formula $\mathsf{Abs}(\psi, \Pi)$ which contains ψ and is representable as a Boolean combination of predicates from Π:

$$\mathsf{Abs}(\psi, \Pi) \;=\; \bigwedge \{\phi \mid \phi \text{ is a Boolean formula over } \Pi \wedge \psi \Rightarrow \phi\}$$

The region $\mathsf{Abs}(\psi, \Pi)$ can be computed by recursively splitting as follows [11]:

$$\mathsf{Abs}(\psi, \Pi) = \begin{cases} \mathit{true} & \text{if } \Pi = \emptyset \text{ and } \psi \text{ satisfiable} \\ \mathit{false} & \text{if } \Pi = \emptyset \text{ and } \psi \text{ unsatisfiable} \\ \quad (p \wedge \mathsf{Abs}(\psi \wedge p, \Pi')) & \\ \qquad\qquad \vee & \text{if } \Pi = \{p\} \cup \Pi' \\ (\neg p \wedge \mathsf{Abs}(\psi \wedge \neg p, \Pi')) & \end{cases}$$

The satisfiability checks can be discharged by a decision procedure [31,13,12]. In the worst case, the computation is exponential in the number of predicates, and several heuristics with better performance in practice have been proposed [34,16].

Using incremental decision procedures, the predicate abstraction $\mathsf{Abs}(\psi, \Pi)$ can be computed as follows. For each predicate $p \in \Pi$, introduce a Boolean variable b_p, and consider the formula

$$\psi \wedge \bigwedge_{p \in \Pi} b_p \leftrightarrow p \tag{2}$$

If the formula is not satisfiable, then the predicate abstraction is false. Otherwise, consider a satisfying assignment to the formula (2), and project the satisfying assignment to the Boolean variables $\{b_p \mid p \in \Pi\}$. By replacing b_p with p in the assignment, we

get a minterm in the predicate abstraction. We can now conjoin the complement of the assignment to the formula (2) and ask for a different satisfying assignment, until there are no more. Incremental decision procedures based on conflict clauses can implement the sequence of queries efficiently. The predicate abstraction is the disjunction of all the satisfying assignments found in this way.

Many implementations of predicate-based software model checkers implement an over-approximation of the predicate abstraction that can be computed efficiently in order to avoid the exponential cost. *Cartesian* predicate abstraction is one such precision-efficiency tradeoff: it can be computed more efficiently than full predicate abstraction but can be quite imprecise in the worst case. Cartesian abstraction formalizes the idea of ignoring relations between components of tuples, and approximates a set of tuples by the smallest Cartesian product containing the set [6]. Formally, the cartesian abstraction of ψ with respect to the set Π of predicates is the smallest (in the implication ordering) region $\mathsf{CartAbs}(\psi, \Pi)$ which contains ψ and is representable as a *conjunction* of predicates from Π. The region $\mathsf{CartAbs}(\psi, \Pi)$ can be computed as:

$$\mathsf{CartAbs}(\psi, \Pi) = \begin{cases} \mathit{true} & \text{if } \Pi = \emptyset \\ p \wedge \mathsf{CartAbs}(\psi, \Pi') & \text{if } \Pi = \{p\} \cup \Pi' \text{ and } (\psi \wedge \neg p) \text{ unsatisfiable} \end{cases}$$

Cartesian predicate abstraction was implemented for C programs as part of SLAM in a tool called c2bp [5], and since then in other software verifiers. While it is sufficient for checking state-machine like properties, it is usually too imprecise in the presence of data structure reasoning.

3.3. Abstraction Refinement

Algorithm AbstractSafety is sound — if it claims ψ is an invariant, then ψ is indeed an invariant — but can produce spurious counterexamples. That is, it can stop with an error even though ψ is an invariant. This can happen if the choice of ϕ in the expansion step is too coarse (so that $\phi \cap \neg\psi$ is satisfiable). As a trivial example, we can always choose true as a candidate ϕ. The idea of an *abstraction refinement* algorithm is to start with some abstraction, and then analyze the counterexamples produced by the abstract model checker to see whether they can be replayed on the concrete system, or if not, to devise a new abstraction that rules out this counterexample (and ideally many more).

We modify the ARG construction in the following way. We augment an ARG with an additional node-labeling function Cex, called the *counterexample labeling*, mapping each node in V to a formula representing, intuitively, the subset of $\Phi(\mathsf{n})$ from which a path to some state not satisfying ψ is possible.

Initially, the ARG consists of two nodes: a root node r with $\Phi(\mathsf{r}) = \mathsf{Init}$ and $\mathsf{Cex}(\mathsf{r}) = \mathit{false}$, and a "top" node top with $\Phi(\mathsf{top}) = \mathit{true}$ and $\mathsf{Cex}(\mathsf{top}) = \neg\psi$.

The construction algorithm has two kinds of steps. First, as before, an *expansion* step adds a-successors of nodes for transitions $a \in A$ (making the graph "more complete"), In addition, a *refinement* step refines the labelings Φ and Cex on nodes (creating new nodes if necessary) to establish, if possible, the condition that all nodes reachable from the root satisfy condition (WL_2) and have a counterexample label false.

The refinement step considers *bad edges* $\mathsf{n} \dot{\rightarrow} \mathsf{n}_1$ in E, for which n is reachable from r, $\mathsf{Cex}(\mathsf{n}) \equiv \mathit{false}$, but $\mathsf{Cex}(\mathsf{n}_1) \not\equiv \mathit{false}$.

EXPAND

$$\frac{\text{pick } \mathsf{n} \in V,\ a \in \mathcal{A},\ \mathsf{r} \xrightarrow{\star} \mathsf{n} \qquad \mathsf{n} \text{ has no outgoing } a\text{-edge in } E \qquad \mathsf{Cex}(\mathsf{n}) = \mathit{false}}{\text{add } \mathsf{n} \xrightarrow{a} \mathsf{top} \text{ to } E}$$

ERROR

$$\frac{\mathsf{Cex}(\mathsf{r}) \neq \mathit{false}}{\textbf{raise Error}}$$

BACK

$$\frac{\text{pick bad edge } \mathsf{n} \xrightarrow{a} \mathsf{n}_1 \text{ in } E \qquad \text{pick formula } \beta \text{ s.t. } \beta \models \Phi(\mathsf{n}) \text{ and } \beta\langle a\rangle\mathsf{Cex}(\mathsf{n}_1)}{\text{update } \mathsf{Cex}(\mathsf{n}) \text{ to } \beta}$$

SWITCH

$$\frac{\begin{array}{c}\text{pick bad edge } \mathsf{n} \xrightarrow{a} \mathsf{n}_1 \text{ in } E,\ \mathsf{n}_2 \in V \\ \mathsf{Cex}(\mathsf{n}_2) \equiv \mathit{false} \qquad \Phi(\mathsf{n}_2) \models \Phi(\mathsf{n}_1) \qquad \Phi(\mathsf{n})\langle a\rangle\Phi(\mathsf{n}_2)\end{array}}{\text{Remove } \mathsf{n} \xrightarrow{a} \mathsf{n}_1 \text{ from } E, \text{ add } \mathsf{n} \xrightarrow{a} \mathsf{n}_2 \text{ to } E}$$

REFINE

$$\frac{\text{pick bad edge } \mathsf{n} \xrightarrow{a} \mathsf{n}_1 \text{ in } E, \text{ formula } \alpha \qquad \Phi(\mathsf{n})\langle a\rangle\alpha \qquad \alpha \models \neg\mathsf{Cex}(\mathsf{n}_1)}{\begin{array}{c}\text{create fresh node } \mathsf{n}_2 \text{ in } V, \text{ set } \Phi(\mathsf{n}_2) = \Phi(\mathsf{n}_1) \wedge \alpha,\ \mathsf{Cex}(\mathsf{n}_2) = \mathit{false} \\ \text{remove } \mathsf{n} \xrightarrow{a} \mathsf{n}_1 \text{ from } E, \text{ add } \mathsf{n} \xrightarrow{a} \mathsf{n}_2 \text{ to } E\end{array}}$$

Figure 1. Rules for a non-deterministic model checking algorithm

We give a non-deterministic description of the algorithm using a set of inference rules shown in Figure 1. The algorithm builds an ARG (V, E, Φ) by application of the inference rules (we omit mentioning the graph explicitly in the rules in Figure 1). The algorithm non-deterministically applies the expansion and refinement steps until they are not applicable or until the root node gets a non-empty counterexample label (i.e., $\mathsf{Cex}(\mathsf{r}) \not\equiv \mathit{false}$).

The expansion step is implemented using the rule (Expand) which expands existing nodes with unexplored transitions. The (Error) rule raises an error if the root node gets a non-empty counterexample label.

The refinement step is implemented by the rules (Switch), (Back), and (Refine). These rules take a bad edge and try to fix the labeling to either establish that all nodes n reachable from r have $\mathsf{Cex}(\mathsf{n}) = \mathit{false}$ or to "push" the counterexamples toward the root.

The applicability of the (Back) rule and the (Switch) and (Refine) rules are complementary. Let $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ be a bad edge, and consider the formula $\Phi(\mathsf{n}) \wedge a \wedge \mathsf{Cex}(\mathsf{n}_1)$. If this formula is satisfiable, then (Back) applies. If not, then either (Switch) or (Refine) —or possibly both— applies.

The (Back) rule propagates a counterexample label $\mathsf{Cex}(\mathsf{n}_1)$ up to its predecessor. If a counterexample can be pushed up to the root, then the (Error) rule raises an error. The (Switch) rule replaces a bad edge out of a node n with a good edge out of n to an existing node. The (Refine) rule adds a new node n_2 to the graph whose label $\Phi(\mathsf{n}_2)$ is stronger than $\Phi(\mathsf{n}_1)$ such that $\mathsf{Cex}(\mathsf{n}_2) \equiv \mathit{false}$. The (Back) and (Refine) rules require the discovery of formulas β and α, respectively.

The algorithm NonDetSafety, on input $\mathcal{S}_\star$ and ψ, non-deterministically applies the rules until either **error** is produced or no rule is applicable or the algorithm goes on forever. During its execution, it builds an ARG (additionally labeled with counterexample labelings).

Theorem 3 1. **[Soundness]** *On input $\mathcal{S}_\star$ and ψ, if Algorithm* NonDetSafety *terminates because no rules are applicable, then ψ is an invariant of $\mathcal{S}_\star$.*

2. **[Validity of Counterexamples]** *If Algorithm* NonDetSafety *terminates with "*error*" then ψ is not an invariant of $\mathcal{S}_\star$.*

PROOF. In the first case, we show that the ARG (V, E, Φ) computed by the algorithm is well-labeled and complete for $\mathcal{S}_\star$ and ψ. We use the invariant that for each edge $\mathsf{n}_1 \xrightarrow{a} \mathsf{n}_2$, if $\mathsf{r} \xrightarrow{\star} \mathsf{r}$ and $\mathsf{Cex}(\mathsf{n}_2) \equiv \mathit{false}$ then $\Phi(\mathsf{n}_1)\langle a\rangle\Phi(\mathsf{n}_2)$. If no rules are applicable, then every node n reachable from r has $\mathsf{Cex}(\mathsf{n}) \equiv \mathit{false}$ (otherwise, either (Back) or (Switch) or (Refine) is applicable). Further, the graph must be complete (otherwise, the (Expand) rule should be applicable at some node).

In the second case, we argue that there is a path from some initial state to a state not in ψ: consider the path in the ARG from the root to top, such that for each node n along the path, $\mathsf{Cex}(\mathsf{n})$ is not empty. By the property of (Back), for each edge $\mathsf{n}_i \xrightarrow{a} \mathsf{n}_{i+1}$ along this path, we have that every state in $\mathsf{Cex}(\mathsf{n}_i)$ has an a-successor to some state in $\mathsf{Cex}(\mathsf{n}_{i+1})$. □

Since invariant verification is undecidable in general, the algorithm is not guaranteed to terminate. For finite-state systems, the algorithm is guaranteed to terminate on each run in which we prioritize (Switch) over (Refine), that is, if we apply (Refine) only when (Switch) is not applicable. We refer to the instance of Algorithm NonDetSafety that prioritizes (Switch) over (Refine) as Algorithm SRSafety.

We now make some concrete choices in the ARG construction, leading to some well-known algorithms.

Implementation: Lazy Abstraction In lazy abstraction [20], the expansion and refinement steps use predicate abstraction in the following way. The algorithm maintains a global set of predicates Π and performs predicate abstraction w.r.t. predicates in Π. It maintains the invariant that the labels of each node can be represented as a Boolean combination of predicates from Π.

For $\mathsf{n} \in V$ and $a \in \mathcal{A}$, the (Expand) and (Switch) steps are combined in the following way. First, the algorithm computes $\mathsf{Abs}(\mathsf{post}(\Phi(\mathsf{n}), a), \Pi)$. If there is already a node $\mathsf{n}_1 \in V$ such that $\mathsf{Abs}(\mathsf{post}(\Phi(\mathsf{n}), a), \Pi) \models \Phi(\mathsf{n}_1)$ and $\mathsf{Cex}(\mathsf{n}_1) \equiv \mathit{false}$, it adds $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ to E. Otherwise, it introduces a new node n_2 and sets $\Phi(\mathsf{n}_2)$ to $\mathsf{Abs}(\mathsf{post}(\Phi(\mathsf{n}), a), \Pi)$ and $\mathsf{Cex}(\mathsf{n}_2)$ to $\mathsf{Abs}(\mathsf{post}(\Phi(\mathsf{n}), a), \Pi) \wedge \neg\psi$.

The β in the (Back) rule is computed as $\mathsf{pre}(\mathsf{Cex}(\mathsf{n}_1), a)$ (without any abstraction).

The (Refine) procedure introduces new predicates to Π through the use of *interpolants*. Let φ_1 and φ_2 be first-order formulas such that $\varphi_1 \wedge \varphi_2$ is unsatisfiable. A formula ψ is called an *interpolant* for (φ_1, φ_2) if (a) $\varphi_1 \Rightarrow \psi$, (b) $\psi \wedge \varphi_2$ is unsatisfiable, and (c) ψ is over the common language of φ_1 and φ_2. Interpolants always exist for first-order logic (extended with recursively enumerable theories), and can be computed from first-order proofs of unsatisfiability (e.g., in a resolution-based proof system).

The (Refine) rule computes an interpolant α' between $\Phi(\mathsf{n}) \wedge a$ and $\neg\mathsf{Cex}(\mathsf{n}_1)'$, and adds all (unprimed) atomic formuals from α' to Π. It then removes the edge $\mathsf{n} \xrightarrow{a} \mathsf{n}_1$ and adds a new node n_2 labeled with $\mathsf{Abs}(\mathsf{post}(\Phi(\mathsf{n}), a), \Pi)$ (with the updated Π) and $\mathsf{Cex}(\mathsf{n}_2) = \mathit{false}$.

The *lazy interpolation* algorithm of McMillan [29] dispenses with the predicate abstraction, and solely uses interpolants in the node labelings. That is, it computes α' as the interpolant between $\Phi(\mathsf{n}) \wedge a$ and $\neg\mathsf{Cex}(\mathsf{n}_1)'$ and uses α as the new label.

Tools based on Abstraction Refinement Several academic and industrial tools have been developed using the ideas of abstraction refinement. We have already mentioned SLAM [4], which pioneered much of the research in the area. SLAM was closely followed by Blast [20], a tool that introduced several ideas such as on-the-fly construction of abstract state spaces and interpolation-based refinement. The tool F-Soft [23] developed at NEC research combined abstraction refinement ideas with bounded model checking.

4. Well-Structured Transition Systems

4.1. A Puzzle with Boxes and Coins

Suppose you are given six boxes, B_1, B_2, ..., B_6, and initially, each box contains one coin. You are allowed two types of operations:

1. Pick a box B_i, $i = 1, \ldots, 5$. Remove a coin from B_i and add two coins to B_{i+1}.
2. Pick a box B_i, $i = 1, \ldots, 4$, remove a coin from B_i and exchange the contents of the boxes B_{i+1} and B_{i+2}.

Exercise 2 *Show that no matter how you apply the two operations, you will eventually terminate, i.e., get to a configuration where you cannot apply any move. [Hint: lexicographic ordering.]*

Exercise 3 *Show that you can get to a configuration in which B_1, ..., B_5 are empty and B_6 has at least $2{\uparrow\uparrow}118$ coins. Here, $a{\uparrow\uparrow}b$ is Knuth's up-arrow notation for iterated exponentiation (or tetration). That is,*

$$a{\uparrow\uparrow}b = \underbrace{a \uparrow (a \uparrow (\ldots \uparrow a) \ldots)}_{b \textit{ times}}$$

where $a \uparrow b$ denotes exponentiation a^b.

We will soon see a general theorem that shows that reachability questions of the kind asked in Exercise 3 are decidable. However, the size of the numbers involved might indicate that decidability would not immediately imply practical algorithms. We will come back to this example in Section 4.5.

4.2. Well-quasi Orderings

A binary relation $\preceq \subseteq S \times S$ over a set S is a *quasi-order* if it is reflexive and transitive. A quasi-order $\preceq$ is a *well-quasi-order* (wqo) if for every infinite sequence

$$s_0 s_1 s_2 \ldots$$

of elements from S, one can always find an i and a j with $i < j$ and $s_i \preceq s_j$.

For example, the $\leq$ relation on natural numbers is a well-quasi ordering. For any finite set S, the equality relation is a well-quasi ordering.

Exercise 4 *Are the following well-quasi orders?* $(\mathbb{Z}, \leq)$*?* $(\mathbb{N}, |)$*, where* $a \mid b$ *iff* a *divides* b*?* $(2^{\mathbb{N}}, \subseteq)$*?*

Proposition 2 $(S, \preceq)$ *is a wqo iff every infinite sequence* $x_1, x_2, \ldots$ *from* S *has an infinite increasing subsequence, i.e., there exist* $i_1, i_2, \ldots$ *such that*

$$x_{i_1} \preceq x_{i_2} \preceq \ldots$$

PROOF. (If) The definition of wqo is weaker than this requirement.

(Only if) Consider the subsequence of all elements x_j such that there is no $x_{j'}$, $j < j'$ with $x_j \preceq x_{j'}$. This subsequence must be finite. Let us say that x_J is the last such element. We construct an infinite increasing sequence as follows. Pick x_{J+1}. Pick x_K such that $x_{J+1} \preceq x_K$. Continue. □

Exercise 5 *(From Schmitz and Schnoebelen) A linear order is a wqo iff it is well-founded. A quasi-order is a wqo iff all linearizations of it are well-founded. (A linearization* $\leq$ *of a qo* $\preceq$ *is a linear order such that* $x \preceq y$ *implies* $x \leq y$*.)*

We can construct new wqos from existing ones.

Proposition 3 *Dickson's Lemma Let* $(S, \preceq_S)$ *and* $(T, \preceq_T)$ *be two wqos. Then* $(S \times T, \preceq)$*, where* $(s, t) \preceq (s', t')$ *iff* $s \preceq_S s'$ *and* $t \preceq_T t'$*, is a wqo.*

As a collorary, consider the set $\mathbb{N}^k$ of k-vectors of natural numbers. The pointwise comparison ordering ($u \leq v$ if for each $i \in \{1, \ldots, k\}$ we have $u_i \leq v_i$) is a well-quasi-order.

Exercise 6 *Let* Σ *be a finite set. A* multiset $\mathbf{m} : \Sigma \to \mathbb{N}$ *maps elements of* σ *to the natural numbers. Define the ordering* $\mathbf{m} \leq \mathbf{m}'$ *iff for each* $\sigma \in \Sigma$*, we have* $\mathbf{m}(\sigma) \leq \mathbf{m}'(\sigma)$*. Show that* $\leq$ *is a wqo. Is* $\leq$ *a wqo if* Σ *is infinite?*

Proposition 4 *Higman's Lemma Let* $(S, \preceq)$ *be a wqo. Then* $(S^*, \preceq_*)$ *is a wqo, where* $s_1 \ldots s_n \preceq_* t_1 \ldots t_m$ *iff*

$$\exists 1 \leq i_1 < i_2 < \ldots < i_n \leq m.\ s_{i_1} \preceq t_{i_1} \wedge \ldots \wedge s_{i_n} \preceq t_{i_n}$$

The ordering $\preceq_*$ is called *subword ordering*. As a special case, the set of strings over a finite alphabet, ordered by subword ordering, is a wqo.

Exercise 7 *We prove Higman's lemma based on a proof by Nash-Williams. Suppose, toward a contradiction, that $(S, \preceq)$ is a wqo, but $(S^*, \preceq_*)$ is not. Then there is some bad sequence over S^* of the form $w_1, w_2, \ldots$ where for each $i < j$, we have $w_i \not\preceq_* w_j$. Of all possible words that start such bad sequences, choose a shortest one. Call this word v_0.*

Now consider all possible bad sequences starting with v_0, and all possible words that follow v_0 in such bad sequences. Pick a shortest one and call it v_1. Repeat the process by choosing, at stage i, a shortest word that continues the sequence $v_0, \ldots, v_{i-1}$ and can be extended to a bad sequence.

1. *Show that the process can be continued forever and yields an infinite sequence $v_0, v_1, \ldots$ which is bad.*
2. *Now write each $v_i = a_i u_i$, where $a_i \in S$ is the first letter of the word v_i, and u_i is the rest of the word. This is always possible, since no bad sequence contains ϵ. Since S is a wqo, we can pick an infinite sequence $a_{i_0} \preceq a_{i_1} \preceq \ldots$ from the sequence $\{a_i\}$.*
   Consider the sequence $u_{i_0}, u_{i_1}, \ldots$ of the suffixes of the bad sequence. Show that if this sequence is good (i.e., satisfies the wqo condition), then the sequence $v_{i_0}, v_{i_1}, \ldots$ is also good.
   Thus, $u_{i_0} u_{i_1} \ldots$ is a bad sequence. Show that this is a contradiction to the choice of $v_0 v_1 \ldots$.
   Conclude that $(S^*, \preceq_*)$ is a wqo.

Exercise 8 *Let $(S, \preceq)$ be a wqo. Let $w_1, w_2, \ldots$ be an infinite sequence of elements from S^* such that $|w_i| < |w_{i+1}|$ for all $i \geq 1$. Show that there exists some $i, j \in \mathbb{N}$, $i < j$, and elements $s_i, s_j^1, s_j^2 \in S$ such that s_i occurs in w_i, s_j^1 and s_j^2 occur in distinct positions in w_j, and $s_i \preceq s_j^1$ and $s_i \preceq s_j^2$.*

4.3. Upward Closure and Finite Bases

Let $(S, \preceq)$ be a wqo. Call a set $U \subseteq S$ *upward closed* if whenever $u \in U$ and $u \preceq v$ we have $v \in U$. For a set $T \subseteq S$, we define the upward closure, $T{\uparrow} = \{s \in S \mid \exists t \in T. t \preceq s\}$.

Proposition 5 *$(S, \preceq)$ is a wqo iff any increasing chain of upward closed sets $U_0 \subseteq U_1 \subseteq \ldots$ eventually stabilize, i.e., there is some k such that $U_k = U_{k+1} = \ldots$.*

Exercise 9 *Prove Proposition 5.*

Upward closed sets can be represented finitely using *basis elements*. An element $s \in U$ is called *minimal* if there is no $t \in U$ distinct from s such that $t \preceq s$. The set of all minimal elements of an upward closed set is called its basis.

Proposition 6 *Every upward closed set has a finite basis.*

PROOF. If not, we can find an infinite sequence of elements that violate the wqo condition. □

Call a set $D \subseteq S$ *downward closed* if whenever $u \in D$ and $v \preceq u$ we have $v \in D$. For a set $T \subseteq S$, we define the downward closure, $T\downarrow = \{s \in S \mid \exists t \in T.s \preceq t\}$.

Exercise 10 *Let Σ be an alphabet and let $\leq$ be a well-quasi-ordering over Σ^*.*

1. *For* any *language $L \subseteq \Sigma^*$, show that $L\uparrow$ and $L\downarrow$ are regular.*
2. *For a language L, show that L is empty iff $L\downarrow$ is empty. Conclude that the construction in part (1) is not effective in general.*

Exercise 11 *Let Σ be a finite alphabet and $\leq$ the subword ordering.*

1. *For a regular language L, give effective constructions for $L\uparrow$ and $L\downarrow$.*
2. *For a context-free language L, give effective constructions for $L\uparrow$ and $L\downarrow$.*
3. *What is the state complexity of computing $L\downarrow$ and $L\uparrow$ for context-free languages L? The best known results show a tight singly exponential bound for $L\uparrow$, and a double exponential upper bound and a single exponential lower bound for $L\downarrow$ [19].*

Exercise 12 *Let Σ be a finite alphabet and let $\preceq$ be defined as $w_1 \preceq w_2$ if $\mathsf{Parikh}(w_1) \leq \mathsf{Parikh}(w_2)$, where $\mathsf{Parikh} : \Sigma^* \to \mathbb{N}^\Sigma$ maps a word to a vector in $\mathbb{N}^\Sigma$ counting the number of occurrences of each letter in w. Show that $\preceq$ is a wqo. For any CFG G, show that an NFA for $L(G)\downarrow$ is at most exponentially larger in $|G|$ and there is a family of CFGs G for which the NFA is $2^{\Omega(|G|)}$.*

4.4. Well-structured Transition Systems (WSTS)

Let $\mathcal{S}$ be a transition system with a well-quasi-ordering $\preceq$ defined on its set of states, and assume that $\mathcal{S}$ has the following monotonicity property: $s \preceq s'$ and $s \to t$ implies there exists a t' such that $t \preceq t'$ and $s' \to t'$. We call such a transition system *well-structured*.

Proposition 7 *Let $\mathcal{S}$ be a monotonic transition system w.r.t. the wqo $\preceq$ and let U be an upward closed set. Then $\mathsf{pre}(U)$ is upward closed.*

If $\mathcal{S}$ is a monotonic transition system with respect to a wqo $\preceq$, and U an upward closed set of states of $\mathcal{S}$, then the sequence of iterations

$$U_0 = U, U_{i+1} = U_i \cup \mathsf{pre}(U_i)$$

stabilizes in a finite number of steps, using Proposition 5. Each set U_i in the sequence is upward-closed, and if the sequence did not terminate, we could construct an infinite sequence of elements violating the well-quasi-ordering assumption on $\preceq$. Thus, backward reachability analysis terminates for any upward closed target set.

The "reachability to an upward closed set" is usually formulated as the *coverability* problem. Given a WSTS $\mathcal{S}$ and two states s and t, the coverability problem asks if there exists a state t' such that $t \preceq t'$ and t' is reachable from t. Clearly, this reduces to checking if $s \in \mathsf{pre}^*(\{t\}\uparrow)$.

In order to get an algorithm out of the backward fixpoint computation, we need to make some effectiveness assumptions. We assume that the relation $\preceq$ is decidable. In addition, we assume effective predecessor computations, usually summarized as an *effective pred-basis* requirement as follows. A WSTS has effective pred-basis if there is a recursive procedure that takes any state s and returns a minimal basis of $\mathsf{pre}(\{s\}\uparrow)$.

Theorem 4 *For a WSTS $\mathcal{S}$ with effective $\preceq$ and effective pred-basis, and an upward closed set U, we can effectively compute a basis for* $\mathsf{pre}^*(U)$.

Even though backward reachability terminates, the bound on the number of iterations of backward reachability can be extremely high (non-primitive recursive).

Consider again the puzzle from Section 4.1. Each state of the system is given by a six-tuple of natural numbers. Both operations are monotonic with respect to $(\mathbb{N}^6, \leq)$. Thus, we can effectively determine if any upward closed set (for example, the one in Exercise 3) can be reached from the initial configuration.

In general, forward analysis for coverability need not terminate even though the backward reachability terminates. For Petri nets, we shall show below that a forward construction of the coverability set does terminate.

The existence of well-quasi-orderings is, in some sense, a canonical requirement for coverability analysis to terminate. For example, from the existence of a bisimulation relation of finite index, one can define a well-quasi-ordering on the state space, and more generally, from the termination of a backward reachability procedure, one can define a suitable well-quasi-ordering on the state space that demonstrates the termination of the reachability analysis.

The power of well-quasi-orderings comes from a large number of *natural* models of computation on which (simple) well-quasi-orderings can be defined [1,15]. Moreover, these well-structured systems satisfy the effectiveness constraints required to design backward reachability algorithms. We now give some examples.

Petri Nets A *Petri net* (PN for short) $N = (S, T, F = \langle I, O \rangle)$ consists of a finite non-empty set S of *places*, a finite set T of *transitions* disjoint from S, and a pair $F = \langle I, O \rangle$ of functions $I \colon T \to \mathbb{M}[S]$ and $O \colon T \to \mathbb{M}[S]$.

To define the semantics of a PN we introduce the definition of *marking*. Given a PN $N = (S, T, F)$, a marking $\mathbf{m} \in \mathbb{M}[S]$ is a multiset which maps each $p \in S$ to a non-negative integer. For a marking $\mathbf{m}$, we say that $\mathbf{m}(p)$ gives the number of *tokens* contained in place p.

A transition $t \in T$ is *enabled at* marking $\mathbf{m}$, written $\mathbf{m}\,[t\rangle$, if $I(t) \preceq \mathbf{m}$. A transition t that is enabled at $\mathbf{m}$ can *fire*, yielding a marking $\mathbf{m}'$ such that $\mathbf{m}' \oplus I(t) = \mathbf{m} \oplus O(t)$. We write this fact as follows: $\mathbf{m}\,[t\rangle\,\mathbf{m}'$. Here, we write $\mathbf{m} \oplus \mathbf{m}'$ for the multiset that maps each $p \in S$ to $\mathbf{m}(p) + \mathbf{m}'(p)$.

Thus, Petri nets define an infinite state transition system, where the states are markings, and there is an edge from $\mathbf{m}$ to $\mathbf{m}'$ labeled t iff $\mathbf{m}\,[t\rangle\,\mathbf{m}'$. Moreover, Petri nets are WSTS, using the natural wqo $\mathbf{m} \leq \mathbf{m}'$ if for each p we have $\mathbf{m}(p) \leq \mathbf{m}'(p)$.

Petri nets are WSTS, but why is this interesting? It turns out that Petri nets are able to model many concurrency constructs in programming languages. Thus, decidability results on Petri nets yields, by reduction, decidability results on these concurrency constructs. We give an example now, and come back to more examples in the next section.

Simple Programs with Dynamic Thread Creation Let us extend our control-flow graphs with a concurrency operation $\mathsf{spawn}(\ell)$. Informally, control-flow graphs with spawn model multi-threaded shared memory programs with dynamic creation of threads. When a spawn operation is performed, a new thread of control is created. The new thread starts executing at location ℓ, and runs in parallel with all existing threads.

Formally, a configuration of the system is a pair $(\mathbf{m}, v)$, where $\mathbf{m}$ is a multiset of locations and v is a valuation to all variables in x. There is a transition $(\mathbf{m}, v) \xrightarrow{a}$

$(\mathbf{m}', v')$ if either (a: multithreaded execution) there is $\ell_1 \in \mathbf{m}$, $(\ell_1, \rho, \ell_2) \in \mathcal{T}$, such that $\mathbf{m}' = \mathbf{m} \ominus \{\ell_1\} \oplus \{\ell_2\}$ and $\rho(v, v')$, or (b: thread creation) there is $\ell_1 \in \mathbf{m}$, $(\ell_1, \mathsf{spawn}(\ell), \ell_2) \in \mathcal{T}$, such that $\mathbf{m}' = \mathbf{m} \ominus \{\ell_1\} \oplus \{\ell_2, \ell\}$ and $v = v'$.

The transition system is infinite-state even when all variables in x range over finite domains. However, in case variables in x are finite-state, we can compute a Petri net that is bisimilar to such a program. Let us assume all variables are Boolean. Informally, the Petri net maintains two places for each variable —one for the value "0" the other for the value "1". Additionally, there is a place for each $\ell \in \mathsf{locs}$. The number of tokens at place ℓ encodes the number of threads in location ℓ. A spawn operation $(\ell_1, \mathsf{spawn}(\ell), \ell_2)$ removes a token from ℓ_1 and adds a token each to ℓ and ℓ_2. A "normal" operation moves a token from ℓ_1 to ℓ_2 and updates the tokens in the variables to reflect the new state.

Exercise 13 *Show how the Petri net can model an assignment* $(\ell, c := a \wedge b, \ell')$ *or a conditional* $(\ell, a \vee b, \ell')$.

The Petri net encoding shows that control state reachability (can a location $\ell \in \mathsf{locs}$ be reached?) is decidable, by reduction to the coverability problem.

Lossy Channel Systems Lossy channel systems consist of parallel compositions of finite-state machines that communicate through sending and receiving messages via a finite set of unbounded lossy FIFO channels. A channel is "FIFO" if the messages are ordered. It is "lossy" if messages can be arbitrarily dropped.

Formally, a Lossy Channel System (LCS) L is a tuple $(S, s_0, C, M, \Sigma, \delta)$, where S is a finite set of (control) states, $s_0 \in S$ is an initial state, C is a finite set of channels, M is a finite set of messages, Σ is a finite set of transition labels, and δ is a finite set of transitions, each of which is of the form (s_1, l, op, s_2), where s_1 and s_2 are states, $l \in \Sigma$, and op is a mapping from C to (send or receive) operations. An operation is either a send operation $!a$, a receive operation $?a$, or an empty operation ϵ, where $a \in M$.

The control states of a system with n finite-state machines is formed as the Cartesian product $S = S_1 \times \ldots \times S_n$ of the control states of each finite-state machine. The initial state of a system with n finite-state machines is a tuple $s_{01}, \ldots, s_{0n}$ of initial states of the components. A configuration (s, w) of L consists of a (global) control state $s \in S$, and a mapping $w : C \to M^*$ giving the contents of each channel. The initial configuration of L is the pair $s_0, \lambda c.\epsilon$.

Channel systems define an infinite labeled transition system, where there is a transition $(s, w) \xrightarrow{l} (s', w')$ iff there is a transition $(s, l, op, s') \in \delta$ and
$w'(c) = w(c) \cdot a$ if $op(c) = !a$, and $w(c) = a \cdot w'(c)$ if $op(c) = ?a$.

In addition, lossiness means that messages can be lost. Let $\preceq_*$ be the subword ordering on M^*. For maps $w, w' : C \to M^*$, we write $w \preceq_* w'$ iff for each $c \in C$, we have $w(c) \preceq_* w'(c)$. We encode lossy transitions by adding the following additional transitions. We add $s, w \xrightarrow{l} s', w'$ iff there are w_1 and w_2 such that $w_1 \preceq_* w$, $w' \preceq_* w_2$, and $s, w_1 \xrightarrow{l} s, w_2$. That is, $s, w \xrightarrow{l} s', w'$ means that s', w' can be reached from s, w by first losing messages from the channels and reaching s, w_1, then performing the transition $s, w_1 \xrightarrow{l} s', w_2$, and finally losing further messages from channels to reach w'.

Exercise 14 1. *The reachability problem for perfect channel systems is undecidable. [Hint: You can encode a Turing machine using a finite state machine and a queue.]*

2. *Show that lossy channel systems are well-structured.*

Lossy channel systems turn out to be a suitable formalism to encode verification problems for *weak memory models* [3].

4.5. Karp Miller Trees

For Petri nets, one can actually compute a finite structure that represents the *coverability set*: the downward closure of the set of reachable markings. The finite structure is called the coverability tree, and is similar to a reachability tree in that nodes represent sets of markings and edges represent firings of transitions. In order to represent downward closed sets exactly, we first introduce a completion $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$ of $\mathbb{N}$ and extend the usual ordering on naturals with $n < \omega$ for all $n \in \mathbb{N}$. Note that $(\mathbb{N}_\omega, \leq)$ is a wqo, and so is $(\mathbb{N}_\omega^k, \leq)$.

A tuple in $\mathbb{N}_\omega^P$ is called an *ω-marking*. Intuitively, an ω in a place is used to indicate that the place can have arbitrarily many tokens. We extend the firing relation to ω-markings, and use $n + \omega = \omega + n = \omega$ for all $n \in \mathbb{N}_\omega$.

The Karp-Miller tree for a Petri net is a rooted directed tree, where nodes are labeled with ω-markings and edges labeled with transitions. The root is labeled with the initial marking $\mathbf{m}_0$. The tree is constructed in the following way.

Suppose we have a node in the tree marked with the ω-marking $\mathbf{m}$, and let $\mathbf{m}_0, \mathbf{m}_1, \ldots, \mathbf{m}_k = \mathbf{m}$ be the sequence of markings from the root to this node. For each transition $t \in T$ such that $\mathbf{m}\,[t\rangle$, we do the following. Let $\mathbf{m}\,[t\rangle\,\mathbf{m}'$.

If $\mathbf{m}' \leq \mathbf{m}_i$ for one of the nodes along the path to the root, we do not add a new node labeled with $\mathbf{m}'$. (Why?)

Otherwise, if $\mathbf{m}' > \mathbf{m}_i$ for some $\mathbf{m}_i$, we build $\mathbf{m}''$ from $\mathbf{m}'$ as follows:

$$\mathbf{m}''(p) = \begin{cases} \omega & \text{if } \mathbf{m}'(p) > \mathbf{m}_i(p) \\ \mathbf{m}'(p) & \text{otherwise} \end{cases}$$

We add a new child to the node labeled with $\mathbf{m}''$, and the edge between them is labeled t.

Otherwise, $\mathbf{m}'$ is incomparable with all $\mathbf{m}_0, \mathbf{m}_k$, and we add a new child to the node labeled with $\mathbf{m}'$, and label the edge t.

Theorem 5 *The tree construction terminates. Let $Cov(N)$ be the set of node labels of the Karp-Miller tree. For any $\mathbf{m}$ reachable from $\mathbf{m}_0$, there is a $\mathbf{m}' \in Cov(N)$ such that $\mathbf{m} \leq \mathbf{m}'$. For any $\mathbf{m} \in Cov(N)$, there is a sequence $\mathbf{m}_1, \mathbf{m}_2, \ldots,$ in $\mathsf{Reach}(N)$ such that $\mathbf{m} = \lim \mathbf{m}_i$.*

The Karp-Miller tree can be used to answer coverability queries: a marking $\mathbf{m}$ is coverable iff there is an ω-marking $\mathbf{m}'$ in $Cov(N)$ such that $\mathbf{m} \leq \mathbf{m}'$. In fact, it can be used to answer other decision problems as well.

Exercise 15 *Termination A Petri net N terminates from an initial marking m_0 if every transition sequence starting from m_0 is finite. A Petri net N is bounded from an initial marking m_0 if there is some $K \in \mathbb{N}$ such that every reachable marking $\mathbf{m}$ satisfies $\mathbf{m} \leq (\lambda p.K)$.*

1. *Show that N does not terminate from m_0 iff there is a run $m_0 [\cdot\rangle \ldots [\cdot\rangle m [\cdot\rangle \ldots [\cdot\rangle m'$ such that $m \leq m'$. How will you use the KM tree to check termination?*
2. *Does termination imply boundedness? Does boundedness imply termination?*
3. *Show that N is bounded from $\mathbf{m}_0$ iff there is only a finite number of markings reachable from $\mathbf{m}_0$.*
4. *How will you use the KM tree to check boundedness?*

How big can the Karp-Miller tree be? The rough answer is "very big." Formally, define the functions

$$A_0(x) = x + 1, \quad A_{n+1}(0) = A_n(1) \quad A_{n+1}(x) = A_n(A_{n+1}(x))$$

and define the Ackermann function $A(n) = A_n(n)$. It is known that the Ackermann function is not primitive recursive.

Theorem 6 *Mayr and Meyer For each $n \in \mathbb{N}$, there is a bounded Petri net N_n of size $O(n)$ that generates $A(n)$ tokens on some place.*

Using this net, the Karp-Miller construction can produce a tree of non-primitive recursive size.

It turns out that an alternate proof of this result can be obtained by generalizing the puzzle from Section 4.1 to n boxes. First, by Exercise 2, we know that the number of coins in any box is bounded along each execution (and all executions terminate). Second, we can show the following transitions (e.g., by induction). First, starting with a configuration with N tokens in one box, one can get to a configuration with 0 tokens in that box and $2N$ tokens in the next, by repeatedly applying operation 1:

$$(N, 0) \xrightarrow{\text{op } 1} (0, 2N)$$

Similarly, when starting with a box with N tokens with two empty boxes to its right, we can put 2^{N+1} tokens in the rightmost box. Begin with

$$(N, 0, 0) \xrightarrow{\text{op } 1} (N-1, 2, 0) \xrightarrow{\text{op } 1} (N-1, 0, 4) \xrightarrow{\text{op } 2} (N-2, 2^2, 0)$$

and show by induction that

$$(N, 0, 0) \xrightarrow{*} (0, 2^N, 0) \xrightarrow{\text{op } 1} (0, 0, 2^{N+1})$$

Continuing in a similar vein, with four boxes, we can get to $2{\uparrow\uparrow}N$:

$$(N, 0, 0, 0) \xrightarrow{*} (0, 2{\uparrow\uparrow}N, 0, 0)$$

and in general, starting with N tokens in the leftmost box of a sequence of n boxes, we can put $2{\uparrow\uparrow}^{n-2}N$ coins in the second box, where $a{\uparrow\uparrow}^n b$ is defined as $a{\uparrow\uparrow}^1 b = a^b$, and for $n > 1$,

$$a{\uparrow\uparrow}^n b = \underbrace{a{\uparrow\uparrow}^{n-1}(a{\uparrow\uparrow}^{n-1}(\ldots{\uparrow\uparrow}^{n-1}a)\ldots)}_{b \text{ times}}$$

Thus, in the puzzle, we can get to very large number of coins, and one can show that the growth of this function is the same as the Ackermann function.

Finally, why is this a Petri net? An initial idea is to have places for boxes and implement the two operations as transitions. While the first operation is easily encoded, notice that the second kind of operation (exchange the contents) is not allowed in a Petri net. However, we can encode the puzzle as a Petri net in the following way. Given n boxes, we add a place for each box (the coins are tokens), and add an additional n^2 places. The idea is that these additional places encode the "name" of a box in unary. There are n places for each box, and we maintain the invariant that exactly one such place has one token and the rest have zero tokens at each point. If the ith place has a token, the corresponding box is now called B_i. We implement exchange by exchanging the names of two boxes. (With a little care, and a lot of Petri net hacking, one can do the same reduction with $O(n \log n)$ additional places.)

4.6. Complexity Bounds

The termination argument based on wqo only guarantees termination, but it does not immediately provide a complexity bound.

For coverability of Petri nets, an upper bound is obtained using an argument of Rackoff, which provides a bound n such that if a marking is coverable, it is coverable in n steps.

Theorem 7 1. *(Rackoff) Let N be a Petri net and $\mathbf{m}$ a marking. $\mathbf{m}$ is coverable from $\mathbf{m}_0$ iff it is coverable by a path of length at most $O((|N| \cdot |\mathbf{m}|)^{2^{|N| \log |N|}})$.*
2. *Coverability for Petri nets is EXPSPACE-complete.*

Membership in EXPSPACE follows from the doubly exponential bound of Rackoff and a non-deterministic log-space procedure for reachability. Hardness is proved in [26], and uses a clever encoding of counter machines where counters are bounded by a doubly exponential function.

The EXPSPACE algorithm is based on a non-deterministic traversal of the state space and is usually not implemented. What about the backward reachability algorithm? Using the Rackoff bound $R(N)$, one can show that backward reachability must terminate in $R(N)$ iterations. Moreover, the size of the constants in the bases computed is then bounded above by $R(N)$ times a parameter dependent on the size of the net. Thus, the backward reachability algorithm can be implemented in doubly exponential time.

Thus, the backward algorithm can be much more efficient than the Karp-Miller construction.

Unfortunately, for other classes of systems, the backward reachability algorithm can be non-primitive recursive.

Theorem 8 *[35] Coverability for lossy channel systems is Ackermann-hard.*

Coverability is also Ackermann-hard for extensions of Petri nets with transfer or reset arcs.

Exercise 16 *Prove Theorem 8. (See [35] for a proof.)*

4.7. EEC and Bounds

The backward reachability algorithm is (almost) optimal but the forward Karp-Miller construction for reachability is very expensive. This is a pity: in many problems, forward algorithms tend to perform better. We now present a very elegant forward algorithm for Petri net coverability (although the technique can be generalized for all WSTS) that combines search with abstraction. Moreover, we show that the asymptotic complexity of the algorithm is again EXPSPACE, so that the worst case complexity is not affected.

The algorithm is called expand, enlarge, and check [18]. It proceeds in rounds. In each round, it computes an under-approximation and an over-approximation of the reachable states. The under-approximations are used to find witnesses in case the target marking is coverable. The over-approximations are used to find a witness in case the target marking is not coverable. Since one of the two situations hold, the algorithm is guaranteed to terminate.

The sequence of under- and over-approximations are as follows. In iteration i, the under-approximation restricts reachability analysis to $\{0, \ldots, i\}^P$. That is, if during the forward reachability analysis, we encounter a marking with some co-ordinate exceeding i, we remove it from consideration. Similarly, the over-approximation performs forward reachability analysis over ω-markings in $\{0, \ldots, i, \omega\}^P$. The under-approximation checks if some marking lower than the target is reachable (return "coverable"). That is, if during the forward reachability analysis, we encounter a marking with some co-ordinate exceeding i, we immediately set that co-ordinate in the marking to ω. The over-approximation checks if the set of abstractly reachable markings is disjoint from the target (return "not coverable"). The correctness of the EEC algorithm shows that the algorithm is sound, complete, and terminating.

What is the complexity of EEC? If the target is coverable, by the Rackoff bound, we know that it must terminate in $R(N)$ iterations. On the other hand, if the target is not coverable, again using the Rackoff bound, we can show that the $R(N)$th over-approximation is precise enough to prove non-coverability. We omit the proof, see [27] for details.

5. Recursive Procedures and Context-Free Reachability

So far, our model of programs has ignored recursion. Transition systems can be used as a model for recursive programs as well, by explicitly encoding the program stack in the state. As the stack can be unbounded, the resulting transition systems are not going to be finite-state in general, even if we interpret the vocabulary over a finite structure, and a "generic" reachability algorithm may not terminate. Instead, we now show how one can get a reachability algorithm by modeling a recursive program as a context-free process (and using algorithms for context free grammars).

Background: Context-free grammars and pushdown automata We recall some concepts from language theory.

A *context-free grammar* (CFG) $G = (\mathcal{X}, \Sigma, \mathcal{P}, \mathcal{X}_0)$ consists of a set $\mathcal{X}$ of non-terminal symbols, a disjoint set Σ of terminal symbols, a set $\mathcal{P} \subseteq \mathcal{X} \times (\mathcal{X} \cup \Sigma)^*$ of production rules, and a starting symbol $\mathcal{X}_0 \in \mathcal{X}$. We write $X \Rightarrow w$ if $(X, w) \in \mathcal{P}$. Given

two strings $u, v \in (\Sigma \cup \mathcal{X})^*$, we define the relation $u \underset{G}{\Rightarrow} v$, if there exists a production $(X, w) \in \mathcal{P}$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\underset{G}{\Rightarrow}^*$ for the reflexive transitive closure of $\underset{G}{\Rightarrow}$. A word $w \in \Sigma^*$ is *recognized* from the non-terminal $X \in \mathcal{X}$ if $X \underset{G}{\Rightarrow}^* w$. We sometimes simply write $\Rightarrow$ instead of $\underset{G}{\Rightarrow}$ if G is clear from the context. We define the language of a CFG G, denoted $L(G)$, as $\{w \in \Sigma^* \mid \mathcal{X}_0 \Rightarrow^* w\}$. A language L is *context-free* (or CFL) if there exists a CFG G such that $L = L(G)$.

A *regular grammar* R is a context-free grammar such that each production is in $\mathcal{X} \times \big((\Sigma \cdot \mathcal{X}) \cup \{\varepsilon\}\big)$. It is known that a language L is *regular* iff $L = L(R)$ for some initialized regular grammar R.

A grammar is a generator of languages. As with regular languages and automata, there are "acceptor" machines for context-free languages. These machines are automata with an auxiliary stack.

A *pushdown automaton* (PDA) $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ consists of a finite set Q of states, an input alphabet Σ, a stack alphabet Γ, a transition function $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$, a start state $q_0 \in Q$, and a set $F \subseteq Q$ of accepting states.

A PDA computes as follows. An input word $w \in \Sigma^*$ is accepted by A if w can be written as $w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\epsilon$, and there exist a sequence of states $r_0, r_1, \dots, r_m$ from Q and a sequence of stack contents $s_0, s_1, \dots, s_m$ from Γ^* such that (1) $r_0 = q_0$ and $s_0 = \epsilon$, that is, the machine starts in the initial state with an empty stack, (2) $r_m \in F$, that is, the machine is in an accepting state at the end, and (3) for each $i \in \{0, \dots, m-1\}$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. The third condition states that the states and the stack contents are updated according to the transition function.

The language of a PDA is the set of all input words accepted by the PDA. It turns out this class is exactly the class of context free languages.

Theorem 9 *For every CFG G there is a pushdown automaton P of size polynomial in G such that $L(G) = L(P)$. Conversely, for each pushdown automaton P, there is a CFG G of size polynomial in P such that $L(P) = L(G)$.*

Control flow graphs with recursion We represent programs using control flow graphs, one for each procedure. The set of procedure names is denoted Σ. For each $\sigma \in \Sigma$, the control flow graph for σ is a labeled, directed graph (V_σ, E_σ), together with a unique entry node $v^e_\sigma \in V_\sigma$ and a unique exit node $v^x_\sigma \in V_\sigma$. We assume the program has (global) variables from some set Y of variables and that each variable $y \in Y$ ranges over a finite domain. Each edge in E_σ is labeled with either a constraint ρ over the free variables $Y \cup Y'$ or a procedure call to a procedure $\sigma' \in \Sigma$ (which can be σ itself). The nodes of the control flow graph correspond to control locations of the program, the entry and exit nodes represent, respectively, where execution of a procedure starts and returns. We assume that each node $v \in V_\sigma$ is reachable from v^e_σ and can reach v^x_σ, and that execution of the program begins at the entry node v^e_{main} of a special procedure $\mathsf{main} \in \Sigma$.

The program representation defines a context-free grammar $G = (V, \Sigma \cup \mathsf{stmts}, \mathcal{P}, v^e_{\mathsf{main}})$, where $V = \uplus\{V_\sigma \mid \sigma \in \Sigma\}$ is the disjoint union of all control flow nodes, stmts is the set of constraints labeling program edges, and the set of productions $\mathcal{P}$ is the smallest set such that

- $(X \to \rho Y) \in \mathcal{P}$ if the edge (X, Y) in the control flow graph is labeled with the operation ρ,
- $(X \to v^e_\sigma Y) \in \mathcal{P}$ if the edge (X, Y) in the control flow graph is labeled with a call to procedure $\sigma \in \Sigma$, and
- $(v^x_\sigma \to \epsilon) \in \mathcal{P}$ for each $\sigma \in \Sigma$.

To capture the effect of constraints on program edges on the program variables, we define an NFA $R = (D, d_0, \mathsf{stmts}, \delta)$ where D is the set of valuations to variables in Y, d_0 is the initial valuation, and $\delta = \{(d, \rho, d') \mid d, d' \in D, (d, d') \models \rho\}$. (For the moment, we omit the final states.)

Intuitively, a leftmost derivation of the grammar G starting from v^e_{main} defines an interprocedurally valid path in the program. A possible global state of the program is given by a state in D obtained by executing the NFA R along the path (note that the constraints can be non-deterministic, and there can be several global states).

We can take a product of the grammar G with the NFA R to construct a grammar G_R in the following way. The grammar $G_R = (V_R, \emptyset, \mathcal{P}_R)$, where $V_R = \{[dvd'] \mid d, d' \in D, v \in V\}$ and $\mathcal{P}_R$ is the least set such that:

- if $(X \to \epsilon) \in \mathcal{P}$ then $([dXd] \to \epsilon) \in \mathcal{P}_R$,
- if $(X \to \rho Y) \in \mathcal{P}$, $(d, \rho, d') \in \delta$, and $d'' \in D$, then $([dXd''] \to [d'Yd'']) \in \mathcal{P}_R$, and
- if $(X \to v^e_\sigma Y) \in \mathcal{P}$, and $d_0, d_1, d_2 \in D$, then $([d_0Xd_2] \to [d_0v^e_\sigma d_1][d_1Yd_2]) \in \mathcal{P}_R$.

The product construction ensures the following invariant: if $[dXd'] \to^* \epsilon$ then there exists $w \in \mathsf{stmts}^*$ such that $d \xrightarrow{w} d'$ in R and $X \to^* w$ in G, and conversely, if $d \xrightarrow{w} d'$ in R and $X \to^* w$ in G then $[dXd'] \to^* \epsilon$ in G_R.

Without loss of generality, we can reduce the invariant verification problem to checking if the program has an execution leading to a special state $d_\star \in D$ when the control location is at v^x_{main}. This reduces to the question if $[dv^e_{\mathsf{main}}d_\star] \to^* \epsilon$, which can be checked using a "marking algorithm" for context-free language emptiness. Moreover, the algorithm can be made symbolic by keeping track of sets of data values, and manipulating them symbolically.

Pushdown reachability Context free grammars and pushdown automata provide two different characterizations for the context free languages. We now present an alternate algorithm for model checking programs with recursion that uses the automaton view.

As a first step, we model programs as *pushdown systems* (PDS), which intuitively are pushdown automata without inputs. A pushdown system $\mathcal{P} = (Q, \Gamma, \delta)$ consists of a finite set Q of control locations, a finite stack alphabet Γ, and a finite set of transition rules $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$. We write $(q, \gamma) \hookrightarrow (q', w)$ if $((q, \gamma), (q', w)) \in \delta$. A configuration of $\mathcal{P}$ is a pair $(q, w) \in Q \times \Gamma^*$.

Let $(q, \gamma) \hookrightarrow (q', w)$. For each $w' \in \Gamma^*$, the configuration $(q, \gamma w')$ can go to (q', ww') in one step. Thus, $(q, \gamma w')$ is an immediate predecessor of (q', ww') and (q', ww') is an immediate successor of $(q, \gamma w')$. We can lift the immediate predecessor relation to sets of configurations in the usual way. Define $\mathsf{pre} : 2^{Q \times \Gamma^*} \to 2^{Q \times \Gamma^*}$ as follows. For a set of configurations C, we have that a configuration $c \in \mathsf{pre}(C)$ iff c is the immediate predecessor of some configuration in C.

We are interested in the reachability question: given two configurations c and c', is c' reachable from c? Note that we cannot simply iterate pre, starting from c', and hope that it converge. Consider for example the PDS with one state q, one stack symbol γ, and a single rule $(q,\gamma) \hookrightarrow (q,\epsilon)$. If we start with (q,ϵ), then $\mathsf{pre}(q,\epsilon) = (q,\gamma)$, $\mathsf{pre}^2(q,\epsilon) = \{(q,\gamma),(q,\gamma\gamma)\}$, and so on.

Instead, we will use a *symbolic representation* using automata. The automata will capture, for a given state q of the PDS, all possible stack configurations w such that there is a way to reach the target configuration starting from (q,w). The use of automata depend on the following result about stack languages of a PDA.

Let $\mathcal{P}$ be a PDA, let c be a configuration, and let q be a state. The *backward stack language* $L(q)$ is defined as the set of all stack configurations w such that (q,w) can reach the configuration c.

Theorem 10 *The backward stack language of a PDA is regular.*

PROOF. Let P be a PDA, and we assume that P accepts on reaching the state q_f with empty stack. For each $q \in Q$, we show the language

$$L(q) = \{w \in \Gamma^* \mid (q,w) \xrightarrow{*} (q_f,\varepsilon)\}$$

is regular. Any computation of P starting with $(q, w_1 \dots w_k)$ can be broken into:

$$(q, w_1 \dots w_k) \xrightarrow{*} (q_1, w_2 \dots w_k) \xrightarrow{*} \cdots \xrightarrow{*} (q_{k-1}, w_k) \xrightarrow{*} (q_f,\varepsilon)$$

where the first symbol is definitely consumed, the second symbol is definitely consumed, and so on. We describe an automaton that has one state for each state of P, i.e., the set of states is Q. The alphabet is Γ. The state q is initial. The state q_f is the only final state. Finally, we define $(q_1,\gamma,q_2) \in \delta$ iff in P, we have

$$(q_1,\gamma) \xrightarrow{*} (q_2,\varepsilon)$$

Note that this check is decidable (in fact, polynomial time, by reduction to emptiness). □

Exercise 17 *The* forward stack language $L_f(q)$ *of a PDA for a state q is defined as the set of all stack configurations w such that (q,w) is reachable from (q_0,ϵ). Show that the forward stack language is regular. [Hint: Consider states $Q \times \Gamma$.]*

We use *multi-automata* to represent sets of configurations. A multi-automaton $A = (Q_A, \Gamma, \delta_A, I_A, F_A)$ for a PDS $\mathcal{P}$ is a finite-state automaton over the stack alphabet Γ, but which has one initial state for each state of the PDS, i.e., I_A is a map from Q to Q_A. The configuration (q,w) is accepted by the multi-automaton A if there is an accepting run of A on w starting from $I_A(q)$. The language $L(A)$ of a multi-automaton is the set of configurations accepted by A. A set of configurations is regular if there is a multi-automaton that recognizes the set.

Now suppose we are given a multi-automaton A. We view it as a (regular) set of configurations, and would like to compute pre^* for this set. Instead of computing pre^* by direct iteration, which need not converge, we construct a sequence $\{Y\}_i$ of regular sets of configurations such that

1. $\mathsf{pre}^i(L(A)) \subseteq Y_i$ for all $i \geq 0$,
2. $Y_i \subseteq \mathsf{pre}^*(L(A))$ for all $i \geq 0$, and
3. there is an i such that $Y_{i+1} = Y_i$.

Properties (1) and (2) ensure that the sequence $\{Y\}_i$ computes $\mathsf{pre}^*(L(A))$, and (3) ensures termination.

We will compute the sequence $\{Y\}_i$ using a saturation procedure on A. That is, the sets Y_i will be accepted by a multi-automaton A_i that has the same set of states as A but possibly more transitions. Since a multi-automaton with n states and alphabet of size m can have at most n^2m transitions, we enforce termination.

Without loss of generality, we assume that no transitions lead into the initial states of A. We start with $Y_0 = L(A)$. Given a multi-automaton A_i that accepts Y_i, we compute A_{i+1} as the automata that has all the transitions in A_i, and adds the following new ones. For every transition rule $(q, \gamma) \hookrightarrow (q', w)$ in δ, and every state r of A_i such that $I_A(q') \xrightarrow{w} r$ in A_i, we add the transition $(I_A(q), \gamma, r)$ to A_{i+1} if it does not exist.

Why is this sound? Note that $(q, \gamma w')$ is an immediate predecessor of (q', ww') by the transition rule. So, for any w', if A_i accepts the word ww' from $I_A(q')$ in A_i, we ensure that in A_{i+1}, the word $\gamma w'$ is accepted from $I_A(q)$.

Theorem 11 *Given a PDS $\mathcal{P}$ and a multi-automaton A for $\mathcal{P}$, there is an effectively constructible multi-automaton $\mathsf{pre}^*(A)$ such that $L(\mathsf{pre}^*(A)) = \mathsf{pre}^*(L(A))$.*

Exercise 18 *What is the complexity of this procedure?*

6. Concurrency

What happens if we combine recursive programs and multithreading? Unfortunately, the analysis of multi-threaded recursive programs communicating through some mechanism is undecidable. The proof uses a reduction from the problem of checking intersection of two context free languages, which is undecidable. Intuitively, each language is represented as a recursive "thread", and there is a common execution to some target state iff there is a common word in their intersection.

Theorem 12 *Reachability analysis for multi-threaded recursive programs is undecidable.*

The undecidability holds (by minor modifications) for most synchronization mechanismsm, such as shared memory, rendezvous, etc. One somewhat surprising decidable case is when the threads communicate solely based on nested locking [24].

6.1. Under-approximation: Context-bounded Reachability

While the general reachability problem is undecidable, a variant of the problem, which computes an *under-approximation* of the reachable states, is decidable. The under-

approximation is called *context-bounded reachability*. Context-bounded reachability takes a recursive multi-threaded program, a target state, and a parameter k, and checks if there is an execution that reaches the target in which the "context" switches from one thread to the other at most k times. For a fixed k, the problem is decidable (even though the problem of finding an execution is undecidable).

Of course, the fact that an under-approximation of the reachable states is decidable is not interesting by itself. (For example, the empty set, a trivial under-approximation, is easily computed.) The interest in context-bounded reachability is the empirical observation that many concurrency errors manifest themselves with low values of k (e.g., 1 or 2). Thus, this particular under-approximation is useful in practice.

We give a proof of decidability of context-bounded reachability in the context of two pushdown automata communicating through shared global variables. Proofs for other settings are similar. The idea is to reduce the problem to the emptiness problem for a single pushdown automaton, which tracks additional state [25].[1]

Let P_1 and P_2 be two pushdown automata. We do not care about the input alphabet, but we assume there is a shared global variable x with values coming from some finite range. A transition of P_1 and P_2 consists of a prior state q, a stack symbol γ, a set of valuations of x (called the guard), a posterior state q', a string of stack symbols w, and a new value for x. Intuitively, when the control is in state q, the popped stack symbol is γ, and the shared variable x is read and if its value satisfies the guard, a new value is atomically written to it, the machine moves to state q' and pushes w on to the stack.

The reachability question asks if some pair of locations (q_{f1}, q_{f2}) of P_1 and P_2 can be reached, starting from some initial pair $(q_{01,}, q_{02})$ (and empty stacks and some default initial value to x).

A run of P_1 and P_2 consists of configurations (q_1, w_1, q_2, w_2, x) of control locations and stack contents of the machines and the current value of x. A run consists of a sequence of configurations, where every consecutive pair is related by the transition relation of P_1 or P_2. A context switch happens in a run if there are two consecutive transitions in the run, the first by P_i and the second by P_{3-i}, for $i \in \{1, 2\}$.

Suppose we bound the total number of context switches in a run by $2K$. We show how to reduce the reachability problem for $2K$-context bounded runs to the emptiness problem for pushdown automata. The intuition is that we "uncouple" the interleaved run of P_1 and P_2 by using non-determinism and additional state. Let us assume P_1 took the first step (there is a symmetric case for P_2, chosen using non-determinism). Instead of one copy of x, the simulating machine keeps $2K$ copies of x, call them $x_1, \ldots, x_{2K}$. The variable x_{2j-1} keeps the value of x at the end of the jth phase of P_1, and x_{2j} contains the value guessed to be the value at the start of the $j+1$th phase of P_1.

The sequential PDA simulates a run with $2K$ context switches by running P_1 with x_1 set to the initial value of x, then non-deterministically deciding at some point that a context switch took place. At that point, it guesses the value at the end of P_2's first phase into x_2 and x_3. It keeps simulating P_1 (the state and the stack does not change by P_2's operations) now using x_3 as the shared variable, and again non-deterministically decides when a context switch took place. At that point, it makes a new guess and stores the guess into x_4 and x_5 and continues. After K such steps, the machine switches to simulating P_2. It starts simulating P_2 starting with x_1 and then guesses that a context switch took place

[1] This proof is different from the original proof by Qadeer and Rehof [33], but is conceptually clearer.

and checks that at that point, the value of x_1 (the current value of the shared state) is the same as the guessed value in x_2. If not, the machine rejects. Otherwise, the machine simulates P_2 using the value stored in x_3 (which represents the shared state when P_1 finished its second phase. At the end of $2K$ phases, the machine has simulated behaviors of all $2K$-bounded executions. Thus, a pair of states is reachable in the original system with at most $2K$ context switches iff it is reachable in the simulating PDA. This latter problem is decidable, of course.

Exercise 19 *Write down the formal construction, and show that the size of the simulating PDA is polynomial in the size of the input for fixed K. Does your construction work if there are n PDAs? Show that the complexity of context-bounded reachability for n PDAs and fixed K is NP-complete.*

6.2. Decidable Models: Asynchronous Programs

By reduction to Petri nets, we showed in Section 4 that safety verification of non-recursive concurrent programs, even in the presence of dynamic allocation of threads, is decidable. On the other hand, in the presence of recursion, just two threads is enough for undecidability. We now look at asynchronous programs, an interesting class in which concurrency and recursion interact in a restricted way and keep the reachability problem decidable.

In an asynchronous program, the programmer can make asynchronous procedure calls which are stored in a task buffer pending for later execution, instead of being executed right away. In addition, the programmer can also make the usual procedure calls where the caller blocks until the callee finishes, and such calls may be recursive. A cooperative scheduler repeatedly picks pending handler instances from the task buffer and executes them atomically to completion. Execution of the handler instance can lead to further handler being posted. The posting of a handler is done using the asynchronous call mechanism. The interleaving of different picks-and-executes of pending handler instances (a pick-and-execute is often referred to as a dispatch) hides latency in the system.

Our formal model consists of three ingredients: a global store of data values, a set of potentially recursive handlers, and a task buffer that maintains a multiset of pending handler instances.

An asynchronous program $\mathsf{AP} = (D, \Sigma, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ consists of a finite set of *global states* D, an alphabet Σ of *handler names*, an alphabet Σ_i of *internal actions* disjoint from Σ, a CFG $G = (\mathcal{X}, \Sigma \cup \Sigma_i, \rightarrow)$, a regular grammar $R = (D, \Sigma \cup \Sigma_i, \delta)$, a multiset $\mathbf{m}_0 \in \mathbb{M}[\Sigma]$ of initial pending handler instances, and an initial state $d_0 \in D$. We assume that for each $\sigma \in \Sigma$, there is a non-terminal $X_\sigma \in \mathcal{X}$ of G.

A *configuration* $(d, \mathbf{m}) \in D \times \mathbb{M}[\Sigma]$ of AP consists of a global state d and a multiset $\mathbf{m}$ of pending handler instances. For a configuration c, we write $c.d$ and $c.\mathbf{m}$ for the global state and the multiset in the configuration respectively. The *initial* configuration c_0 of AP is given by $c_0.d = d_0$ and $c_0.\mathbf{m} = \mathbf{m}_0$.

The semantics of an asynchronous program is given as a labeled transition system over the set of configurations, with a transition relation $\rightarrow \subseteq (D \times \mathbb{M}[\Sigma]) \times \Sigma \times (D \times \mathbb{M}[\Sigma])$ defined as follows: let $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$, $d, d' \in D$ and $\sigma \in \Sigma$

$$(d, \mathbf{m} \oplus \mathbb{M}[\sigma]) \xrightarrow{\sigma} (d', \mathbf{m} \oplus \mathbf{m}')$$

iff

$$\exists w \in (\Sigma \cup \Sigma_i)^* \colon d \underset{R}{\Rightarrow}^* w \cdot d' \wedge X_\sigma \underset{G}{\Rightarrow}^* w \wedge \mathbf{m}' = \mathsf{Parikh}_\Sigma(w) \ .$$

Intuitively, we model the (potentially recursive) code of a handler using a context-free grammar. The code of a handler does two things: first, it can change the global state (through R), and second, it can add new pending handler instances (through derivation of a word in Σ^*). Together, the transition relation $\rightarrow$ states that there is a transition from configuration $(d, \mathbf{m} \oplus \mathbb{M}[\sigma])$ to $(d', \mathbf{m} \oplus \mathbf{m}')$ if there is an execution of handler σ that changes the global state from d to d' and adds to the task buffer the handler instances given by $\mathbf{m}'$. Note that the multiset $\mathbf{m}$ (the current content of the task buffer minus the pending handler instance σ) is unchanged while σ executes, and that the order in which the handler instances are added to the task buffer is immaterial (hence, in our definition, we take the Parikh image of w).

Finally, we conclude from the definition of their semantics that asynchronous programs satisfy the following form of *monotonicity*. Let us first define the ordering $\sqsubseteq\sqsubseteq (D \times \mathbb{M}[\Sigma]) \times (D \times \mathbb{M}[\Sigma])$ such that $c \sqsubseteq c'$ iff $c.d = c'.d \wedge c.\mathbf{m} \preceq c'.\mathbf{m}$. Also we have:

$$\forall \sigma \in \Sigma \, \forall c_1 \, \forall c_2 \, \forall c_3 \, \exists c_4 \colon c_1 \xrightarrow{\sigma} c_2 \wedge c_1 \sqsubseteq c_3 \text{ implies } c_3 \xrightarrow{\sigma} c_4 \wedge c_2 \sqsubseteq c_4 \ .$$

Therefore, the transitions system $\big((D \times \mathbb{M}[\Sigma], \sqsubseteq), \rightarrow, c_0\big)$ defined by asynchronous programs are *well-structured transition systems*.

A *run* of an asynchronous program is a finite or infinite sequence

$$c_0 \xrightarrow{\sigma_0} c_1 \cdots c_k \xrightarrow{\sigma_k} c_{k+1} \cdots$$

of configurations c_i starting from the initial configuration c_0. A configuration c is *reachable* if there is a finite run $c_0 \xrightarrow{\sigma_0} \cdots \xrightarrow{\sigma_{k-1}} c_k$ with $c_k = c$.

The *global state reachability* problem for an asynchronous program takes as input an asynchronous program and a global state $d_f \in D$, and asks if there is a reachable configuration c such that $c.d = d_f$.

The key insight in the analysis of asynchronous programs is that the effect of a handler is only to add tasks to the task buffer, and it does not matter in what order tasks got added to the buffer. That is, we do not have to reason precisely about the context-free language of posts of tasks by a handler, we only need to look at the numbers of different tasks that were posted by the handler.

Let us define a function $\mathsf{Parikh} : \Sigma^* \rightarrow \mathbb{N}^\Sigma$ such that $\mathsf{Parikh}(w)(a)$ is the number of occurrences of the letter a in the word w. For example, $\mathsf{Parikh}(aabacc)(a) = 3$, $\mathsf{Parikh}(aabacc)(b) = 1$, etc. We extend Parikh to languages in the natural way: $\mathsf{Parikh}(L) = \{\mathsf{Parikh}(w) \mid w \in L\}$. It turns out that the image of the map Parikh has a simple structure [32]:

Theorem 13 *[Parikh's Theorem] For every context free language L, there is an (effectively computed) regular language L' such that* $\mathsf{Parikh}(L) = \mathsf{Parikh}(L')$.

What Parikh's theorem allows us to do is to replace the pushdown automaton of a handler by a finite automaton that is equivalent w.r.t. its effect on the task buffer: the stack is gone! Once this transformation is done, we can convert an asynchronous program (now without recursion in the handlers) to a Petri net, roughly as follows. There is a place for each control location of each handler, representing the control location of the currently executing handler, and a place for each value of the global variable. Additionally, there is a place representing the scheduler, and a place for each task that tracks how many instances of that task are currently pending (the "task buffer" for that task). To model a handler call, for each task, there is a transition that consumes a token from the scheudler and one token from the task buffer and produces a token at the start location of the handler for the task. The post of a task puts a token in its task buffer. When the handler returns, the token is returned from its control location to the scheduler, so that a new handler can be chosen for execution. Together, this gives a reduction from asynchronous programs to Petri nets, and shows that global state reachability is decidable by reducing the question to a coverability question on the Petri net.

Unfortunately, the regular language guaranteed by Parikh's theorem may only be represented by non-deterministic finite automata that have size exponential in the grammar. Thus, the above reduction gives a doubly exponential space algorithm.

Exercise 20 *Consider the singleton language $L_n = \{a^{2^n}\}$, for $n \in \mathbb{N}$. Show that for each n, there is a CFG of size $O(n)$ for L_n but every NFA for L_n has $2^{O(n)}$ states.*

Exercise 21 *(See Ganty and Majumdar [17].) Show that a Petri net can be simulated by a recursion-free asynchronous program. Thus, the global state reachability question for asynchronous programs is EXPSPACE-hard.*

With a little more care, one can show a stronger reduction: a Petri net that is *polynomial* in the size of the asynchronous program.

Lemma 1 *[17] For every asynchronous program AP, there is a Petri net $N(\mathsf{AP})$ of size polynomial in AP such that (1) $(d, \mathbf{m})$ is reachable in AP iff $\mathbb{M}[d] \oplus \mathbf{m}$ is reachable in $N(\mathsf{AP})$, and (2) d is reachable for some $\mathbf{m}$ iff $\mathbb{M}[d]$ is coverable in $N(\mathsf{AP})$.*

The crux of the polynomial-time reduction is a representation of the Parikh image of a context-free language as a Petri net that is size polynomial in the grammar. Intuitively, the Petri net for the Parikh image of a context-free grammar has places corresponding to the terminals and non-terminals of the grammar. A production $A \to BC$ (respectively, $A \to a$ of the grammar consumes a token from the place A and puts one token each in the places B and C (respectively, one token in a). Suppose we start with a marking which has exactly one token in the start non-terminal S. If we reach a marking in which there are no tokens in any place corresponding to non-terminals, the tokens in the places corresponding to terminals gives the Parikh image of some word in the language. Conversely, the Parikh image of every word in the language can be obtained in this way.

Unfortunately, there is one technicality here: Petri nets cannot test a place for emptiness, so how can we ensure that all non-terminal places are zero? For this, we use a result from [14] on *index bounded* languages.

Let G be a CFG. For $k \geq 1$, we define the sub-relation $\underset{G}{\Rightarrow}[k]$ of $\underset{G}{\Rightarrow}$ as $u \underset{G}{\Rightarrow}[k]v$ iff $u \underset{G}{\Rightarrow} v$ and both u and v contain at most k occurrences of non-terminals. The *k-index*

language of G is $L^{(k)}(G) = \{w \in \Sigma^* \mid S \underset{G}{\Rightarrow} [k]^* w\}$. Intuitively, the k-index language contains those strings in $L(G)$ which can be derived by a sequence which never has more than k non-terminals in any intermediate sentence.

Lemma 2 *[14] For every CFG G with n non-terminals,* $\mathsf{Parikh}(L(G)) = \mathsf{Parikh}(L^n(G))$.

Now we can construct a Petri net. Intuitively, the Petri net keeps a "store" of n tokens, and maintains the invariant that the sum of all tokens in the non-terminal places together with the remaining tokens in the store is exactly n. Thus, if at any point, the store has n tokens, we can conclude that all non-terminal places are empty. So, the "handler return" will be modeled by checking that the store has n tokens, and this is a check that the Petri net can perform.

Let $G = (V, \Sigma, \to, S)$ be a CFG with n non-terminals. Without loss of generality, we assume that each rule in G is of the form $A \to BC$ or $A \to a$. Consider the Petri net $(V \cup \Sigma \cup \{\mathsf{store}\}, T, \langle I, O\rangle)$, where T is the smallest set containing a transition t for each rule $A \to BC$ with $I(t) = \{A, \mathsf{store}\}$, $O(t) = \{B, C\}$, and a transition t for each rule $A \to a$ with $I(t) = \{A\}$ and $O(t) = \{a, \mathsf{store}\}$. Intuitively, we consume a token from the store whenever we increase the number of tokens in non-terminal places, and give back a token whenever a non-terminal is reduced to a terminal.

Let m_0 be the marking with one token in place S and $n - 1$ tokens in place store. Then, any marking reachable from m_0 in which there are n tokens in the store corresponds to a derivation of some word in $L^n(G)$, and the tokens in the terminal places correspond to the Parikh image of that word. Further, all words in the Parikh image can be obtained in this way. This construction can be combined with the previous reduction (in case there was no recursion) to get a polynomial-sized Petri net.

Theorem 14 *[17] Global state reachability of asynchronous programs is EXPSPACE-complete.*

References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society Press, 1996.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[4] T. Ball, V. Levin, and S. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Languages Design and Implementation*, pages 203–213. ACM, 2001.

[6] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 2031, pages 268–283. Springer-Verlag, 2001.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 99: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1579, pages 193–207. Springer-Verlag, 1999.

[8] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[11] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer-Aided Verification*, Lecture Notes in Computer Science 1633, pages 160–171. Springer-Verlag, 1999.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 4963, pages 337–340. Springer-Verlag, 2008.

[13] B. Dutertre and L. de Moura. Yices SMT solver. `http://yices.csl.sri.com/`.

[14] J. Esparza, P. Ganty, S. Kiefer, and M. Luttenberger. Parikh's theorem: A simple and direct automaton construction. *Information Processing Letters*, 111:614–619, 2011.

[15] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere. Technical Report LSV-98-4, Laboratoire Spécification et Vérification, 1998.

[16] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 02: Principles of Programming Languages*, pages 191–202. ACM, 2002.

[17] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6, 2012.

[18] G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check: New algorithms for the coverability problem of wsts. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.

[19] H. Gruber, M. Holzer, and M. Kutrib. More on the size of higman-haines sets: Effective constructions. *Fundam. Inform.*, 91(1):105–121, 2009.

[20] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[21] T. Henzinger, R. Majumdar, and J.-F. Raskin. A classification of symbolic transition systems. *ACM Transactions on Computational Logic*, 6:1–32, 2005.

[22] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[23] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[24] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL 07: Principles of Programming Languages*, pages 303–314, 2007.

[25] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[26] R. Lipton. The reachability problem is exponential-space hard. Technical Report 62, Department of Computer Science, Yale University, 1976.

[27] R. Majumdar and Z. Wang. Expand, enlarge, and check for branching vector addition systems. In *CONCUR 2013: Concurrency Theory*, Lecture Notes in Computer Science 8052, pages 152–166. Springer, 2013.

[28] K. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[29] K. L. McMillan. Lazy abstraction with interpolants. In *CAV 2006*, Lecture Notes in Computer Science, pages 123–136. Springer-Verlag, 2006.

[30] M. Minsky. *Finite and infinite machines*. Prentice-Hall, 1967.

[31] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[32] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[33] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[34] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, Lecture Notes in Computer Science 1633, pages 443–454. Springer-Verlag, 1999.

[35] P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS 10: Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 6281, pages 616–628. Springer, 2010.