Analysis of Recursively Parallel Programs

AHMED BOUAJJANI and MICHAEL EMMI, LIAFA, Université Paris Diderot

10

We propose a general formal model of isolated hierarchical parallel computations, and identify several fragments to match the concurrency constructs present in real-world programming languages such as Cilk and X10. By associating fundamental formal models (vector addition systems with recursive transitions) to each fragment, we provide a common platform for exposing the relative difficulties of algorithmic reasoning. For each case we measure the complexity of deciding state reachability for finite-data recursive programs, and propose algorithms for the decidable cases. The complexities which include PTIME, NP, EXPSPACE, and 2EXPTIME contrast with undecidable state reachability for recursive multithreaded programs.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; F.1.2 [**Models of Computation**]: Parallelism and concurrency

General Terms: Algorithms, Reliability, Verification

Additional Key Words and Phrases: Concurrency, parallelism, verification

ACM Reference Format:
Bouajjani, A. and Emmi, M. 2013. Analysis of recursively parallel programs. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 10 (November 2013), 49 pages.
DOI: http://dx.doi.org/10.1145/2518188

1. INTRODUCTION

Despite the ever-increasing importance of concurrent software (e.g., for designing reactive applications, or parallelizing computation across multiple processor cores), concurrent programming and concurrent program analysis remain challenging endeavors. The most widely available facility for designing concurrent applications is *multithreading*, where concurrently executing sequential threads nondeterministically interleave their accesses to a shared memory. Such nondeterminism leads to unexpected interthread interference, causing elusive "Heisenbugs" [Gray 1985] which are notoriously difficult to reproduce and repair. To prevent such bugs, programmers are faced with the difficult task of predicting and preventing undesirable interleaving, for example, by employing lock-based synchronization, without preventing the benign interleaving necessary to achieve reactivity or parallelism.

This work concerns the automatic prediction of programming errors via the "static determination of dynamic properties of programs" [Cousot and Cousot 1976], that is, computing soundly and statically which from a given set of conditions must always hold at a given set of program locations.

M. Emmi was supported by a postdoctoral fellowship from the Fondation Sciences Mathématiques de Paris. This work was partially supported by the project ANR-09-SEGI-016 Veridye.
Authors' address: A. Bouajjani and M. Emmi (corresponding author), LIAFA, Université Paris Diderot, France; email: mje@liafa.univ-paris-diderot.fr.

DOI: http://dx.doi.org/10.1145/2518188

Automatic approaches, such as *dataflow analysis* [Nielson et al. 1999; Reps et al. 1995; Sharir and Pnueli 1981] and *model checking* [Clarke and Emerson 1981; Clarke et al. 2001; Queille and Sifakis 1982] typically reason abstractly over finite quotients of program configurations, abstracting data values into a finite number of equivalence classes.[1] For example, to determine a given program property, an analysis might need only to differentiate between three sets of values $(-\omega, -1], [0], [1, \omega) \subseteq \mathbb{Z}$, which an integer-valued program variable may store at a given program location, rather than differentiating between every possible value [Cousot and Cousot 1977]; similarly, an analysis might consider only valuations to a given set of predicates, rather than the program variable values themselves [Graf and Saïdi 1997], for example, whether an integer-valued variable can be even or odd, or whether two program variables can be equal or unequal. Timely termination of such analyses relies either on these quotients being small or *well structured* [Abdulla et al. 1996; Finkel and Schnoebelen 2001], or on techniques to force the convergence of fixed point calculations [Cousot and Cousot 1977].

In this work we assume a program model in which variables store only values from finite domains. Such a *finite-value* model is sufficient to capture any finite quotient of data valuations, and thus corresponds to the usual abstractions leveraged by dataflow analyses and model checkers.

Automatic static analysis of multithreaded programs is notoriously expensive, and this is largely due to the underlying concurrency model of thread interleaving: to be sound, analyses must assume that any thread can interfere at any moment during the execution of another, interleaving its own shared-memory accesses. For finite-value programs, the complexity of computing state reachability—that is, whether a given program location can be encountered in some execution—is PSPACE-complete[2] with a statically fixed number of threads that do not call recursive procedures [Kozen 1977]; this complexity rises to EXPSPACE-complete when an unbounded number of threads can be dynamically created: the combined results of Lipton [1976] and Rackoff [1978] show EXPSPACE-completeness of the coverability problem for vector addition systems, and German and Sistla [1992], Ball et al. [2001], and Delzanno et al. [2002] demonstrate the connection between multithreaded programs and vector addition systems. State reachability becomes undecidable when (as few as two) threads can call recursive procedures [Ramalingam 2000], or when an unbounded number of threads can distinguish themselves with unique identifiers [Apt and Kozen 1986; Clarke et al. 2001]. Existing approaches to analysis cope with these complexities either by underapproximating the set of program behaviors, considering relatively few thread interleavings [Emmi et al. 2011; Esparza and Ganty 2011; Kahlon 2009; Qadeer and Rehof 2005], or by exploring a coarse overapproximation via abstraction [Elmas et al. 2009; Flanagan and Qadeer 2003; Henzinger et al. 2003].

In order to avoid the intricacies arising from thread interleaving which is implicit in the syntax of multithreaded programs, many experts are advocating *explicitly parallel* programming languages [Lee 2006], and several such industrial-strength languages have been developed [Allen et al. 2006; Burckhardt et al. 2010; Chamberlain et al. 2007; Charles et al. 2005; Fatahalian et al. 2006; Halstead 1985; Leijen et al. 2009; Randall 1998; Segulja and Abdelrahman 2011; Sen and Viswanathan 2006]. Such systems introduce various mechanisms for creating (e.g., **fork**, **spawn**, **post**) and

[1] Cousot and Cousot's [1977] framework does not require abstractions to be quotients, nor finite.

[2] In order to isolate concurrent complexity from the exponential factor in the number of program variables, we consider throughout this work a fixed number of variables in each procedure frame, unifying the concept of a "program" with its transition system. This allows us a PTIME point-of-reference for state reachability in recursive sequential programs [Chaudhuri 2008; Reps et al. 1995].

State-Reachability in Recursively Parallel Programs			
	section	complexity	language/feature
Task-Passing			
general	Sec. 4	undecidable	futures (Sect. 8.1), revisions (Sect. 8.2)
Single-Wait			
non-aliasing	Sec. 9.1	PTIME	futures (Sect. 8.1)†, revisions (Sect. 8.2)†
local scope	Sec. 9.2	EXPSPACE	—
global scope	Sec. 9.3	EXPSPACE	asynchronous programs (Sect. 8.3)
general	Sec. 9.4	2EXPTIME	—
† For programs without bidirectional task-passing.			
Multi-Wait (single region)			
local scope	Sec. 12.1	NP	Cilk (Sect. 11.1), parallel for-each (Sect. 11.3)
general	Sec. 12.2	decidable	async (X10, Sect. 11.2)

Fig. 1. Summary of results for computing state reachability for finite-value recursively parallel programs.

consuming (e.g., **join**, **sync**) concurrent computations, and either encourage (through recommended programming practices) or ensure (through static analyses or runtime systems) that parallel computations execute in isolation without interference from others, through data partitioning [Charles et al. 2005], data replication [Burckhardt et al. 2010], functional programming [Halstead 1985], message passing [Pratikakis et al. 2011], or version-based memory access models [Segulja and Abdelrahman 2011], perhaps falling back on transactional mechanisms [Larus and Rajwar 2006] when complete isolation is impractical. Although some of these systems behave deterministically, consuming one concurrent computation at a time, many are sensitive to the order in which multiple isolated computations are consumed. Furthermore, some permit computations creating an unbounded number of subcomputations, returning to their superiors an unbounded number of handles to unconsumed computations. Even without multithreaded interleaving, nondeterminism in the order in which an unbounded number of computations are consumed has the potential to make program reasoning complex.

In this work we investigate key questions on the analysis of interleaving-free programming models. Specifically, we ask to what extent such models simplify automated reasoning, how these models compare with each other, and how to design appropriate analysis algorithms. Our attempt to answer these questions, summarized in Figure 1, is outlined as follows.

Section 3 We introduce a general interleaving-free parallel programming model on which to express the features found in popular parallel programming languages.

Section 4 We uncover a surprisingly complex feature appearing in some existing languages, including Multilisp's futures [Halstead 1985] and Burckhardt et al.'s [2010] revisions: the ability to pass concurrent subcomputations both to and from other subcomputations—which we refer to as *(bidirectional) task passing*—leads to undecidability of state reachability.

Section 7 We demonstrate an encoding of programs without (bidirectional) task passing into a fundamental formal model of computation: vector addition systems with recursive transitions and zero-test transitions (RVASS+Z).

Section 8 We show that the concurrency features present in many real-world programming languages such as Multilisp can be captured in the "single-wait" fragment of our general model, in which only a single subcomputation must complete before a waiting computation continues.

Section 9 We measure the complexity of computing state reachability for various naturally occurring subfragments of single-wait programs, by restricting aliasing or the scope of subcomputations, and provide asymptotically optimal state-reachability algorithms.

Section 11 We show that the concurrency features present in many other real-world programming languages such as Cilk and X10 (modulo the possibility of interleaving) can be captured in the "multi-wait" fragment of our general model, in which all subcomputations must complete before a waiting computation continues.

Section 12 We measure the complexity of computing state reachability for various naturally occurring subfragments of multi-wait programs, and provide asymptotically optimal state-reachability algorithms.

Generally speaking, reasoning for the "single-wait" fragment of Section 8 is less difficult than for the "multi-wait" fragment of Section 11, and we demonstrate a range of complexities from PTIME, NP, EXPSPACE, and 2EXPTIME for various scoping restrictions in Sections 9 and 12. Despite these worst-case complexities, a promising line of work has already demonstrated effective algorithms for practically occurring EXPSPACE-complete state-reachability problem instances based on simultaneously computing iterative under- and overapproximations, and rapidly converging to a fixed point [Geeraerts et al. 2006; Jhala and Majumdar 2007].

As mentioned, our focus on state reachability in finite-data programs reflects the usual settings of dataflow analysis and model checking, which ask whether a given property can be violated in a given program, under an analysis-dependent, often finite, data abstraction. Our focus on programs without inter-thread interference reflects an increasingly fashionable design choice, either language-wide, such as in the case of Multilisp futures [Halstead 1985] and Burckhardt et al.'s [2010] revisions, partially enforced, such as in Cilk [Randall 1998], or widely practiced, for example, in X10 [Charles et al. 2005]. Applying our algorithms in practice may rely on data abstraction [Cousot and Cousot 1977; Graf and Saïdi 1997], and separately ensuring isolation statically [Elmas et al. 2009] or dynamically [Larus and Rajwar 2006], or on approximating possible interleaving by overapproximation [Flanagan and Qadeer 2003] or underapproximation [Qadeer and Rehof 2005]; still, our handling of computation-order nondeterminism is precise.

We thus present a classification of concurrency constructs, connecting programming language features to fundamental formal models which highlight the sources of concurrent complexity resulting from each feature, and provide a platform for comparing the difficulty of formal reasoning in each. We hope that these results may be used both to guide the design of impactful program analyses, as well as to guide the design and choice of languages appropriate for various programming problems.

2. PRELIMINARIES

To begin with we introduce the background concepts and notation to be used throughout. See Sipser's [1997] textbook for complete definitions of the more ubiquitous concepts, which are abridged in the following.

A Σ-*word* is a finite sequence $w \in \Sigma^*$ of symbols from an *alphabet* Σ; the symbol ε denotes the empty word, and a *language* $L \subseteq \Sigma^*$ is a set of words.

The *Parikh-image* $\Pi(w)$ of a word $w \in \Sigma^*$ is the multiset $m \in \mathbb{M}[\Sigma]$, or equivalently the vector $\vec{n} \in \mathbb{N}^{|\Sigma|}$, such that for each $a \in \Sigma$, $m(a)$, respectively, $\vec{n}(a)$ is the number of occurrences of a in w. The *Parikh-image* of a language $L \subseteq \Sigma^*$ is the set of Parikh-images of each constituent word: $\Pi(L) = \{\Pi(w) : w \in L\}$. Two languages L_1 and L_2 are *Parikh-equivalent* when $\Pi(L_1) = \Pi(L_2)$.

A *Finite-State Automaton (FSA)* $\mathcal{A} = \langle Q, \Sigma, \hookrightarrow \rangle$ over an alphabet Σ is a finite set Q of *states*, along with a set $\hookrightarrow \subseteq Q \times \Sigma \times Q$ of *transitions*. Given initial and accepting states $q_0, q_f \in Q$, the language $\mathcal{A}(q_0, q_f)$ is the set of Σ-words labeling runs of $\mathcal{A}$ which begin in the initial state q_0 and terminate in the accepting state q_f. The *language-emptiness problem* for finite-state automata is to decide, given an automaton $\mathcal{A}$ and states $q_0, q_f \in Q$, whether $\mathcal{A}(q_0, q_f) = \emptyset$.

A *Context-Free Grammar (CFG)* $\mathcal{G} = \langle V, \Sigma, \hookrightarrow \rangle$ over an alphabet Σ is a finite set V of *variables*, along with a finite set $\hookrightarrow \subseteq V \times (V \cup \Sigma)^*$ of *productions*. Given an initial variable $v_0 \in V$, the language $\mathcal{G}(v_0)$ is the set of Σ-words derived by $\mathcal{G}$ from the initial variable v_0.

A *PushDown Automaton (PDA)* $\mathcal{A} = \langle Q, \Sigma, \Gamma, \hookrightarrow \rangle$ over an alphabet Σ is a finite set Q of *states*, along with a *stack alphabet* Γ, and a finite set $\hookrightarrow \subseteq Q \times \Gamma \times \Sigma \times \Gamma^* \times Q$ of *transitions*. A configuration qw is a state $q \in Q$ paired with a stack-symbol sequence $w \in \Gamma^*$. Given initial and accepting states $q_0, q_f \in Q$, the language $\mathcal{A}(q_0, q_f)$ is the set of Σ-words labeling runs of $\mathcal{A}$ which begin in the initial configuration $q_0\varepsilon$ and terminate in an accepting configuration $q_f w$, for some $w \in \Gamma^*$. The *language-emptiness problem* for pushdown automata is to decide, given an automaton $\mathcal{A}$ and states $q_0, q_f \in Q$, whether $\mathcal{A}(q_0, q_f) = \emptyset$.

A *vector addition system (VASS) [Karp and Miller 1969]* $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ of dimension $k \in \mathbb{N}$ is a finite set Q of *states*, along with a finite set $\hookrightarrow \subset Q \times \mathbb{N}^k \times \mathbb{N}^k \times Q$ of *transitions*. A *configuration* $q\vec{n}$ is a state $q \in Q$ paired with a vector $\vec{n} \in \mathbb{N}^k$, and a *step* from $q_1\vec{n}_1$ to $q_2\vec{n}_2$ is a transition $q_1 \xhookrightarrow{\vec{n}\text{-}\vec{n}_+} q_2$ between states q_1 and q_2, subtracting $\vec{n}$- from $\vec{n}_1$, and adding $\vec{n}_+$, that is, such that $\vec{n}_1 \geq \vec{n}$- and $\vec{n}_2 = \vec{n}_1 \ominus \vec{n}\text{-} \oplus \vec{n}_+$; a *run* is a sequence of configurations, each from which there exists a step to its successor. Given initial and accepting states $q_0, q_f \in Q$, $\mathcal{A}(q_0, q_f)$ is the set of vectors $N_f \subseteq \mathbb{N}^k$ such that $\mathcal{A}$ has a run which begins in $q_0\mathbf{0}$ and terminates in $q_f\vec{n}_f$, for some $\vec{n}_f \in N_f$. The *state-reachability problem* (respectively, the *(configuration-, or vector-) reachability problem*) for vector addition systems is to decide, given a system $\mathcal{A}$ and states $q_0, q_f \in Q$ (respectively, and a vector $\vec{n}_f \in \mathbb{N}^k$), whether $\mathcal{A}(q_0, q_f) \neq \emptyset$ (respectively, whether $\vec{n}_f \in \mathcal{A}(q_0, q_f)$).

A *Turing Machine (TM)* $\mathcal{A} = \langle Q, \Sigma, \hookrightarrow \rangle$ over an alphabet Σ is a finite set Q of *states*, along with a finite set $\hookrightarrow \subseteq Q \times \Sigma \times \{\mathrm{L}, \mathrm{R}\} \times \Sigma \times Q$ of *transitions*. A configuration $\langle q, w_1, w_2 \rangle$ is a state $q \in Q$ along with two words $w_1, w_2 \in \Sigma^*$. Given initial and accepting states $q_0, q_f \in Q$, the language $\mathcal{A}(q_0, q_f)$ is the set of Σ-words w such that $\mathcal{A}$ has a run which begins in an initial configuration $\langle q_0, \varepsilon, w \rangle$ and terminates in an accepting configuration $\langle q_f, w_1, w_2 \rangle$, for some and $w_1, w_2 \in \Sigma^*$. The *language-emptiness problem* for Turing machines is to decide, given a machine $\mathcal{A}$ and states $q_0, q_f \in Q$, whether $\mathcal{A}(q_0, q_f) = \emptyset$.

3. RECURSIVELY PARALLEL PROGRAMS

We consider a simple concurrent programming model where computations are hierarchically divided into isolated parallely executing tasks. Each task executes sequentially while maintaining *regions* (i.e., containers) of *handles* to other tasks. The initial

$$
\begin{array}{rcl}
P & ::= & (\ \textbf{proc}\ p\ (\textbf{var}\ \mathtt{l}\colon T)\ s\ )^* \\
s & ::= & s;\ s \ \mid\ \mathtt{l} := e \ \mid\ \textbf{skip} \ \mid\ \textbf{assume}\ e \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \ \mid\ \textbf{while}\ e\ \textbf{do}\ s \\
 & \mid & \textbf{call}\ \mathtt{l} := p\ e \ \mid\ \textbf{return}\ e \\
 & \mid & \textbf{post}\ r \leftarrow p\ e\ \vec{r}\ d \ \mid\ \textbf{ewait}\ r \ \mid\ \textbf{await}\ r
\end{array}
$$

Fig. 2. The grammar of recursively parallel programs. Here T is an unspecified type, p ranges over procedure names, e over expressions, r over regions, and d over return-value handlers.

task begins without task handles. When a task t creates a subordinate (child) task u, t stores the handle to u in one of its regions, at which point t and u begin to execute in parallel. The task u may then recursively create additional parallel tasks, storing their handles in its own regions. At some later point when t requires the result computed by u, t must *await* the completion of u—that is, blocking until u has finished—at which point t consumes its handle to u. When u does complete, the value it returns is combined with the current state of t via a programmer-supplied *return-value handler*. In addition to creating and consuming subordinate tasks, tasks can transfer ownership of their subordinate tasks to newly created tasks—by initially passing to the child a subset of task handles—and to their superiors upon completion—by finally passing to the parent all unconsumed tasks.

This model permits arbitrarily concurrent executions. Each task along with all the tasks it has created execute completely in parallel. As tasks can create tasks recursively, the total number of concurrently executing tasks has no bound, even when the number of handles stored by each task is bounded.

3.1. Program Syntax

Let Procs be a set of procedure names, Vals a set of values, Exprs a set of expressions, Regs a finite set of region identifiers, and Rets $\subseteq$ (Vals $\to$ Stmts) a set of return-value handlers. The grammar of Figure 2 describes our language of *recursively parallel programs*. We intentionally leave the syntax of expressions e unspecified, though we do insist Vals contains true and false, and Exprs contains Vals and the *(nullary) choice operator* $\star$. We refer to the class of programs restricted to a finite set of values as *finite-value* programs, and to the class of programs restricted to at most $n \in$ Nats (respectively, 1) region identifiers as *n-region* (respectively, *single-region*) programs. A *sequential program* is a program without **post**, **ewait**, and **await** statements.

Each program P declares a sequence of procedures named $p_0 \ldots p_i \in \mathsf{Procs}^*$, each p having single type-T parameter $\mathtt{l}$ and a top-level statement denoted s_p; as statements are built inductively by composition with control-flow statements, s_p describes the entire body of p. The set of program statements s is denoted Stmts.

Remark 3.1. The programming language we consider is simple yet expressive, since the syntax of types and expressions is left free, and we lose no generality by considering only a single parameter variable per procedure; multiple procedure parameters, as well as procedure-local variables, can be encoded as parts of the procedure parameter. Taking the semantics that procedure-local variables begin uninitialized with nondeterministically chosen values, we suppose procedure **call**s and **post**s systematically pass nondeterministically chosen values for the local variable parts of the procedure argument, using the $\star$ operator.

Intuitively, a $\textbf{post}\ r \leftarrow p\ e\ \vec{r}\ d$ statement stores the handle to a newly created task executing procedure p in the region r; besides the procedure argument e, the newly created task is passed the subset of the parent's task handles in regions $\vec{r}$, and a

```
proc fib (var n: ℕ)
  reg r
  var sum: ℕ := 0
  if n < 2 then
      return 1
  else
      post r ← fib (n-1) ε
        (λv. sum := sum + v);
      post r ← fib (n-2) ε
        (λv. sum := sum + v);
      await r;
      return sum
```

Fig. 3. A recursively parallel Fibonacci function.

```
proc main ()
    reg r
    ...
    post r ← periodic-task () ε d;
    while true do ewait

proc periodic-task ()
    reg r
    ...
    // repost self
    post r ← periodic-task () ε d;
    return
```

Fig. 4. A recursively parallel periodic task handler.

return-value handler d. The **ewait** r statement blocks execution until *some* task whose handle is stored in region r completes, at which point its return-value handler is executed. Similarly, the **await** r statement blocks execution until *all* tasks whose handles are stored in region r complete, at which point all of their return-value handlers are executed, in some order. We refer to the **call**, **return**, **post**, **ewait**, and **await** as *inter-procedural statements*, and the others as *intra-procedural statements*, and insist that return-value handlers are comprised only of intra-procedural statements. The **assume** e statement proceeds only when e evaluates to **true**; we use this statement in subsequent sections to block undesired executions in our encodings of other parallel programming models.

Example 3.2. Figure 3 illustrates an implementation of the Fibonacci function as a single-region recursively parallel program. Alternate implementations are possible, for example, by replacing the **await** statement by two **ewait** statements, or storing the handles to the recursive calls in separate regions. Note that in this implementation task handles are not passed to child tasks (ε specifies the empty region sequence) nor to parent tasks (all handles are consumed by the **await** statement before returning).

Figure 4 illustrates an implementation of a periodic task handler. Each time the periodic task executes, it reposts another instance of itself, which is stored in region `r`, and passed back to the `main` procedure upon return, whose **ewait** statement eventually consumes the task, and reexecutes it.

3.2. Parallel Semantics

Unlike recursive sequential programs, whose semantics is defined over *stacks* of procedure frames, the semantics of recursively parallel programs is defined over *trees* of procedure frames. Intuitively, the frame of each posted task becomes a child of the posting task's frame. Each step of execution proceeds either by making a single intra-procedural step of some frame in the tree, by creating a new frame for a newly posted task, or by removing the frame of a completed task; unconsumed subtask frames of a completed task are added as children to the completed task's parent.

A *task* $t = \langle \ell, s, d \rangle$ is a valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable `l`, along with a statement s to be executed, and a return-value handler $d \in \mathsf{Rets}$. (Here s describes the entire body of a procedure p that remains to be executed, and is initially set to p's top-level statement s_p.) A *tree configuration* c is a finite unordered tree of task-labeled vertices and region-labeled edges, and the set of configurations is denoted $\mathsf{Configs}$. Let $\mathbb{M}[\mathsf{Configs}]$ denote the set of configuration multisets. We represent configurations inductively, writing $\langle t, m \rangle$ for the tree with t-labeled root whose child subtrees are given by a *region valuation* $m : \mathsf{Regs} \to \mathbb{M}[\mathsf{Configs}]$: for $r \in \mathsf{Regs}$, the multiset $m(r)$ specifies

$$\textsc{Skip}\quad T[\mathbf{skip};\ s] \xrightarrow[P]{\text{seq}} T[s] \qquad \textsc{Assume}\quad \frac{\mathbf{true} \in e(T)}{T[\mathbf{assume}\ e] \xrightarrow[P]{\text{seq}} T[\mathbf{skip}]} \qquad \textsc{Assign}\quad \frac{\ell' \in e(\ell)}{\langle \ell, S[\mathtt{1}\ \mathtt{:=}\ e], d\rangle \xrightarrow[P]{\text{seq}} \langle \ell', S[\mathbf{skip}], d\rangle}$$

$$\textsc{If-Then}\quad \frac{\mathbf{true} \in e(T)}{T[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \xrightarrow[P]{\text{seq}} T[s_1]} \qquad \textsc{If-Else}\quad \frac{\mathbf{false} \in e(T)}{T[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \xrightarrow[P]{\text{seq}} T[s_2]}$$

$$\textsc{Loop-Do}\quad \frac{\mathbf{true} \in e(T)}{T[\mathbf{while}\ e\ \mathbf{do}\ s] \xrightarrow[P]{\text{seq}} T[s; \mathbf{while}\ e\ \mathbf{do}\ s]} \qquad \textsc{Loop-End}\quad \frac{\mathbf{false} \in e(T)}{T[\mathbf{while}\ e\ \mathbf{do}\ s] \xrightarrow[P]{\text{seq}} T[\mathbf{skip}]}$$

Fig. 5. The intra-procedural transition relation for recursively parallel programs.

the collection of subtrees connected to the root of $\langle t, m\rangle$ by an r-edge. The *initial region valuation* $m_\emptyset$ is defined by $m_\emptyset(r) \stackrel{\text{def}}{=} \emptyset$ for all $r \in \mathsf{Regs}$. The singleton region valuation $(r \mapsto c)$ maps r to $\{c\}$, and $r' \in \mathsf{Regs} \setminus \{r\}$ to $\emptyset$, and the union $m_1 \cup m_2$ of region valuations is defined by the multiset union of each valuation: $(m_1 \cup m_2)(r) \stackrel{\text{def}}{=} m_1(r) \cup m_2(r)$ for all $r \in \mathsf{Regs}$. The projection $m\,|\,\vec{r}$ of a region valuation m to a region sequence $\vec{r}$ is defined by $(m\,|\,\vec{r})(r') = m(r')$ when r' occurs in $\vec{r}$, and $(m\,|\,\vec{r})(r') = \emptyset$ otherwise.

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. For convenience, we define

$$e(\langle \ell, s, d\rangle) \stackrel{\text{def}}{=} e(\ell) \stackrel{\text{def}}{=} [\![e[\,\ell/\mathtt{1}]]\!]_{\mathrm{e}}$$

—as $\mathtt{1}$ is the only variable, the expression $e[\,\ell/\mathtt{1}]$ has no free variables.

To reduce clutter and focus on the relevant parts of transition rules in the program semantics, we introduce a notion of contexts. A *configuration context* C is a tree with a single $\diamond$-labeled leaf, task-labeled vertices and leaves otherwise, and region-labeled edges. We write $C[c]$ for the configuration obtained by substituting a configuration c for the unique $\diamond$-labeled leaf of C. We use configuration contexts to isolate individual task transitions, writing, for instance $C[\langle t, m\rangle] \to C[\langle t', m\rangle]$ to indicate an intra-procedural transition of the task t. Similarly a *statement context* $S = \diamond;\ s_1;\ \ldots;\ s_i$ is a $\diamond$-led sequence of statements, and we write $S[s_0]$ for the statement obtained by substituting a statement s_0 for the unique occurrence of $\diamond$ as the first symbol of S, indicating that s_0 is the next-to-be-executed statement. A *task-statement context* $T = \langle \ell, S, d\rangle$ is a task with a statement context S in place of a statement, and we write $T[s]$ to indicate that s is the next statement to be executed in the task $\langle \ell, S[s], d\rangle$. Finally, we write $C[\langle T[s_1], m\rangle] \to C[\langle T[s_2], m'\rangle]$ to denote a transition of a task executing a statement s_1 and replacing s_1 by s_2—normally s_2 is the **skip** statement. Since the current statement s of a task $T[s]$ does not affect expression evaluation, we liberally write $e(T)$ to denote the evaluation $e(T[s])$.

We say a task $t = \langle \ell, S[s], d\rangle$ is *completed* when its next-to-be-executed statement s is **return** e, in which case we define $\mathsf{rvh}(t) \stackrel{\text{def}}{=} \{d(v) : v \in e(\ell)\}$ as the set of possible return-value handler statements for t; $\mathsf{rvh}(t)$ is undefined when t is not completed.

Figure 5 and Figure 6 define the transition relation $\to^{rpp/p}$ of recursively parallel programs as a set of operational steps on configurations. The intra-procedural transitions $\to^{seq}$ of individual tasks in Figure 5 are standard. More interesting are the inter-procedural transitions of Figure 6. The POST-T rule creates a procedure frame to execute in parallel, and links it to the current frame by the given region, passing

$$\textsc{Intra-T}\ \frac{t_1 \xrightarrow[P]{\text{seq}} t_2}{C[\langle t_1, m\rangle] \xrightarrow[P]{\text{rpp/p}} C[\langle t_2, m\rangle]} \qquad \textsc{Post-T}\ \frac{v \in e(T) \qquad m' = m \setminus (m\,|\,\vec{r}) \cup \big(r \mapsto \langle\langle v, s_p, d\rangle, m\,|\,\vec{r}\rangle\big)}{C[\langle T[\mathbf{post}\ r \leftarrow p\ e\ \vec{r}\ d], m\rangle] \xrightarrow[P]{\text{rpp/p}} C[\langle T[\mathbf{skip}], m'\rangle]}$$

$$\exists\textsc{wait-T}\ \frac{m_1 = \big(r \mapsto \langle t_2, m_2\rangle\big) \cup m_1' \qquad s \in \mathsf{rvh}(t_2)}{C[\langle T_1[\mathbf{ewait}\ r], m_1\rangle] \xrightarrow[P]{\text{rpp/p}} C[\langle T_1[s], m_1' \cup m_2\rangle]} \qquad \forall\textsc{wait-Done-T}\ \frac{m(r) = \emptyset}{C[\langle T[\mathbf{await}\ r], m\rangle] \xrightarrow[P]{\text{rpp/p}} C[\langle T[\mathbf{skip}], m\rangle]}$$

$$\forall\textsc{wait-Next-T}\ \frac{m_1 = \big(r \mapsto \langle t_2, m_2\rangle\big) \cup m_1' \qquad s \in \mathsf{rvh}(t_2)}{C[\langle T_1[\mathbf{await}\ r], m_1\rangle] \xrightarrow[P]{\text{rpp/p}} C[\langle T_1[s;\ \mathbf{await}\ r], m_1' \cup m_2\rangle]}$$

Fig. 6. The tree-based transition relation for parallely executing recursively parallel programs with task passing. As the choice of configuration context C is left open, every enabled transition of *every task* in a tree configuration is enabled.

ownership of tasks in the specified region sequence to the newly created frame. The ∃WAIT-T rule consumes the result of a single child frame in the given region, and applies the return-value handler to update the parent frame's local valuation. Similarly, the ∀AWAIT-NEXT-T and ∀AWAIT-DONE-T rules consume the results of every child frame in the given region, applying their return handlers in the order they are consumed. The semantics of **call** statements reduces to that of **post** and **ewait**: supposing an unused region identifier $\mathtt{r}_{call}$, we translate each statement **call** `l := p e` into the sequence

$$\mathbf{post}\ \mathtt{r}_{call} \leftarrow p\ e\ \varepsilon\ \mathtt{d}_{call};\ \mathbf{ewait}\ \mathtt{r}_{call} \qquad \text{where } \mathtt{d}_{call}(v) \stackrel{\text{def}}{=} \mathtt{l := } v$$

and where ε denotes an empty sequence of region identifiers.

A *parallel execution of a program P (from c_0 to c_j)* is a configuration sequence $c_0 c_1 \ldots c_j$ where $c_i \rightarrow^{rpp/p} c_{i+1}$ for $0 \le i < j$. An initial condition $\iota = \langle p_0, \ell_0\rangle$ is a procedure $p_0 \in \mathsf{Procs}$ along with a value $\ell_0 \in \mathsf{Vals}$. A configuration $\langle\langle \ell_0, s, d\rangle, m_\emptyset\rangle$ is called $\langle p_0, \ell_0\rangle$-*initial* when s is the top-level statement of p_0. A configuration c_f is called ℓ_f-*final* when there exists a context C such that $c_f = C[\langle t, m\rangle]$ and $\mathtt{l}(t) = \ell_f$. We say a valuation ℓ is *reachable in P* from ι when there exists an execution of P from some c_0 to c_f, where c_0 is ι-initial and c_f is ℓ-final.

Problem 1 (*State-Reachability*). The *state-reachability problem* is to determine, given an initial condition ι of a program P and a valuation ℓ, whether ℓ is reachable in P from ι.

3.3. Sequential Semantics

Since tasks only exchange values at creation and completion time, the order in which concurrently executing tasks make execution steps does not affect computed program values. In this section we leverage this fact and focus on a particular execution order in which at any moment only a single task is enabled. When the currently enabled task encounters an **ewait**/**await** statement, suspending execution to wait for a subordinate task t, t becomes the currently enabled task; when t completes, control returns to its waiting parent. At any moment only the tasks along one path ρ in the

$$\textsc{Intra-S}\ \frac{t_1 \xrightarrow[P]{\text{seq}} t_2}{\langle t_1, m\rangle c \xrightarrow[P]{\text{rpp/s}} \langle t_2, m\rangle c} \qquad \textsc{Post-S}\ \frac{v \in e(T) \qquad m' = m \setminus (m\,|\,\vec{r}) \cup \big(r \mapsto \langle\!\langle v, s_p, d\rangle\!\rangle, m\,|\,\vec{r}\big)}{\langle T[\mathbf{post}\ r \leftarrow p\ e\ \vec{r}\ d], m\rangle c \xrightarrow[P]{\text{rpp/s}} \langle T[\mathbf{skip}], m'\rangle c}$$

$$\exists\textsc{wait-S}\ \frac{m = \big(r \mapsto c_0\big) \cup m'}{\langle T[\mathbf{ewait}\ r], m\rangle c \xrightarrow[P]{\text{rpp/s}} c_0\langle T[\mathbf{skip}], m'\rangle c} \qquad \textsc{Return-S}\ \frac{s \in \mathsf{rvh}(t_1)}{\langle t_1, m_1\rangle\langle T_2[\mathbf{skip}], m_2\rangle c \xrightarrow[P]{\text{rpp/s}} \langle T_2[s], m_1 \cup m_2\rangle c}$$

$$\forall\textsc{wait-Done-S}\ \frac{m(r) = \emptyset}{\langle T[\mathbf{await}\ r], m\rangle c \xrightarrow[P]{\text{rpp/s}} \langle T[\mathbf{skip}], m\rangle c} \qquad \forall\textsc{wait-Next-S}\ \frac{m = \big(r \mapsto c_0\big) \cup m'}{\langle T[\mathbf{await}\ r], m\rangle c \xrightarrow[P]{\text{rpp/s}} c_0\langle T[\mathbf{skip};\ \ \mathbf{await}\ r], m'\rangle c}$$

Fig. 7. The stack-based transition relation for sequentially executing recursively parallel programs with task passing. Unlike the tree-based transition relation of Figure 6, only the enabled transitions of *a single task*—that the task of the topmost frame on the stack—are enabled in any given configuration.

configuration tree have ever been enabled, and all but the last task in ρ are waiting for their child in ρ to complete. We encode this execution order into an equivalent stack-based operational semantics, which essentially transforms recursively parallel programs into sequential programs with an unbounded auxiliary storage device used to store subordinate tasks. We interpret the **ewait** and **await** statements as procedure calls which compute the values returned by previously posted tasks.

We define a *frame* to be a configuration in the sense of the tree-based semantics of Section 3.2, that is, a finite unordered tree of task-labeled vertices and region-labeled edges. (Here all nonroot nodes in the tree are posted tasks that have yet to take a single step of execution.) In our stack-based semantics, a *stack configuration* c is a sequence of frames, representing a procedure activation stack.

Figures 5 and 7 define the sequential transition relation $\rightarrow^{rpp/s}$ of recursively parallel programs as a set of operational steps on configurations. Interesting here are the rules for **ewait** and **await**. The $\exists$WAIT-S rule blocks the currently executing frame to obtain the result for a single, nondeterministically chosen frame c_0 in the given region, by pushing c_0 onto the activation stack. Similarly, the $\forall$WAIT-NEXT-S and $\forall$WAIT-DONE-S rules block the currently executing frame to obtain the results for every task in the given region, in a nondeterministically chosen order. Finally, the RETURN-S applies a completed task's return-value handler to update the parent frame's local valuation. The definitions of *sequential execution*, *initial*, and *reachable* are nearly identical to their parallel counterparts.

LEMMA 3.3. *The parallel semantics and the sequential semantics are indistinguishable with respect to state reachability, that is, for all initial conditions ι of a program P, the valuation ℓ is reachable in P from ι by a parallel execution if and only if ℓ is reachable in P from ι by a sequential execution.*

PROOF. We demonstrate the correspondence between stack and tree configurations by defining a map f from stack configurations to tree configurations. For this, we suppose each task $\langle t, m\rangle$ waiting on a task in region r, that is, whose last-executed statement was either **ewait** r or **await** r, is annotated by the region r on which it waits. We define f inductively, by $f(\langle t, m\rangle) \stackrel{\text{def}}{=} \langle t, m\rangle$, and $f(\langle t_2, m_2\rangle c)$ adds to the tree $f(c)$ an r-edge from $\langle t_1, m_1\rangle$ to $\langle t_2, m_2\rangle$, where $\langle t_1, m_1\rangle$ is the unique node of $f(c)$ which is both a leaf,

and waiting, on a task in region r; f is well defined since $f(c)$ always has a unique leaf which is waiting on another task, according to the sequential semantics.

Given f, it is routine to verify that each sequential execution $c_0c_1 \dots c_j$ corresponds to a valid parallel execution $f(c_0)f(c_1) \dots f(c_j)$. In the other direction, it can be shown that any parallel execution can be reordered into a parallel execution $c_0c_1 \dots c_j$ such that $f^{-1}(c_0)f^{-1}(c_1) \dots f^{-1}(c_j)$ is a well-defined sequential execution, since the steps of any two tasks unrelated by a given tree configuration commute in parallel executions. □

4. COMPUTING STATE-REACHABILITY WITH TASK-PASSING

Recursively parallel programs allow pending tasks to be passed *bidirectionally*: both from completed tasks and to newly created tasks. As we demonstrate, this capability makes the state-reachability problem undecidable—even for the very simple cases of recursive programs with at least one region, and nonrecursive programs with at least two regions. (We show afterwards that state reachability remains decidable for the somewhat degenerate case of nonrecursive programs with only one region.) Essentially, when pending tasks can be passed to newly created tasks, it becomes possible to construct and manipulate unbounded task chains by keeping a handle to the most recently created task, after having passed the handle of the previously most recently created task to the most recently created task. Such unbounded chains of pending tasks can be used to simulate an arbitrary unbounded and ordered storage device.

Definition 4.1 (*Task Passing*). A program which contains a statement $\textbf{post}\, r \leftarrow p\ e\ \vec{r}\ d$, such that $|\vec{r}| > 0$ is called *task passing*.

Example 4.2. Figure 8 illustrates two quicksort-based algorithms, implemented as recursively parallel programs, with and without task passing. As they are written, the task-passing version allows more parallelism, since the sorting of each array partition need not wait for the other's `filter` to complete.

Note that even in nontask-passing programs, any pending, unconsumed tasks upon the completion of a procedure are transferred to the corresponding region containers of the awaiting procedure frame.

4.1. State Reachability is Undecidable for Task-Passing Programs

The *task depth* of a program P is the maximum length of a sequence $p_1 \dots p_i$ of procedures in P such that each p_j contains a statement $\textbf{post}\, r \leftarrow p_{j+1}\ e\ \vec{r}\ , d$ for $0 < j < i$, and some $r \in \mathsf{Regs}$, $e \in \mathsf{Exprs}$, $\vec{r} \in \mathsf{Regs}^*$, and $d \in \mathsf{Rets}$. Programs with unbounded task depth are *recursive*, and are otherwise *nonrecrusive*.

THEOREM 4.3. *The state-reachability problem for n-region finite-value task-passing parallel programs is undecidable for (a) nonrecursive programs with $n > 1$, and (b) recursive programs with $n > 0$.*

We prove Theorem 4.3 by two separate reductions from the language emptiness problem for Turing machines to "single-wait" programs, that is, those using **ewait** statements but not **await** statements (as in Section 8). In essence, as each task handle can point to an unbounded chain of task handles, we can construct an unbounded Turing machine tape by using one task chain to store the contents of cells to the left of the tape head, and another chain to store the contents of cells to the right of the tape head. If only one region is granted but recursion is allowed (i.e., as in (b)), we can still construct the tape using the task chain for the cells right of the tape head, while using the (unbounded) procedure stack to store the cells left of the head. We prove (a)

```
proc sort( reg r1 )
    reg r2
    var a1, a2: [ℕ]
    var i: ℕ

    // the argument array is returned
    // by the task in region r1, whose
    // return-value handler d1 will
    // copy the array reference to a1.
    ewait r1;

    if (length(a1) < 2)
      return a1;

    i := a1[0];
    post r1 ← filter(a1,λx.x <i) ε d1;
    post r2 ← filter(a1,λx.x ≥i) ε d1;

    // task-passing happens here
    post r1 ← sort() r1 d1;
    post r2 ← sort() r2 d2;
    ewait r1; // result in a1
    ewait r2; // result in a2
    return append(a1,a2)

where d1(a) ≝ a1 := a
      d2(a) ≝ a2 := a
```

```
proc sort( var a1: [ℕ] )
    reg r1, r2
    var a2: [ℕ]
    var i: ℕ

    if (length(a1) < 2)
      return a1;

    i := a1[0];
    post r1 ← filter(a1,λx.x <i) ε d1;
    post r2 ← filter(a1,λx.x ≥i) ε d2;
    ewait r1; // result in a1
    ewait r2; // result in a2

    post r1 ← sort(a1) ε d1;
    post r2 ← sort(a2) ε d2;
    ewait r1; // result in a1
    ewait r2; // result in a2
    return append(a1,a2)

where d1(a) ≝ a1 := a
      d2(a) ≝ a2 := a
```

Fig. 8. Two recursively parallel sorting algorithms: (LEFT) with task passing, and (RIGHT) without. While the non-task-passing `sort` procedure is called with the original array as an argument, the task-passing `sort` procedure must be called asynchronously, with the **post** statement, and passed a region argument containing a single task which returns the original array; its return-value handler $\mathtt{d}_1$ assigns the array into `a1`.

and (b) separately, both by reduction from the language emptiness problem for Turing machines.

PROOF (a). By reduction from the language emptiness problem for Turing machines, let $\mathcal{A} = \langle Q, \Sigma, \hookrightarrow \rangle$ be a Turing machine with transitions $\hookrightarrow = \{d_1, \ldots, d_j\}$, and let $q_0, q_f \in Q$. We assume, without loss of generality, that upon entering the accepting state q_f, the machine $\mathcal{A}$ performs a sequence of left moves until reaching the end of the tape; that is, $\langle q_f, a, \mathrm{L}, a, q_f \rangle \in \hookrightarrow$ for all $a \in \Sigma$. For any fixed $a_0 \in \Sigma$, we define a task-passing program $P_{\mathcal{A}}$, in Figure 9, with two regions $\mathtt{r}_L$ and $\mathtt{r}_R$, along with an initial `main` procedure, an auxiliary nonrecursive procedure `p`, and a return-value handler `d` which assigns the return value to the `sym` variable of the `main` procedure.

Essentially, $P_{\mathcal{A}}$ simulates the tape of $\mathcal{A}$ using two singly-linked lists of frames of `p`; the $\mathtt{r}_L$-linked list stores tape cells to the left of the current cell, whose symbol is stored in the variable `sym`, and the $\mathtt{r}_R$-linked list stores tape cells to the right. The `main` procedure first fills the input tape cells with nondeterministically chosen values, then enters a loop in which each iteration simulates one $\mathcal{A}$-transition. $P_{\mathcal{A}}$ simulates left moves by adding a cell to the head of the $\mathtt{r}_R$-linked list, and right moves by adding a cell to the head of the $\mathtt{r}_L$-linked list. At each step, the return-value handler `d` applied at an **ewait** statement updates `main`'s `sym` variable with the symbol stored in the target tape cell. Indeed, the sole purpose of procedure `p` is to store a symbol, and a pointer to a next tape cell.

```
proc main ()
    reg r_L, r_R
    var state: Q
    var sym: Σ
    var done: 𝔹 := false

    // guess the contents
    // of a valid tape input
    while ⋆ do
        post r_R ← p ⋆ r_R d;
    state := q_0;
    sym := ⋆;

    while ⋆ do
      // take a step of A

    // check: q_f reachable?
    done := true;
    sym := a_0;
    return
```

```
// code for taking a
// step of A
if ⋆ then s_1
else if ⋆ then s_2
else if ..
else s_j;

// the auxiliary procedure
proc p (reg r, var sym: Σ)
  return sym

// the return-value handler
d(a): sym := a
```

```
// statement s_i for
// left transitions
// d_i : q --a/b,L--> q'
assume state = q;
assume sym = a;
post r_R ← p b r_R d;

state := q';
// overwrites sym
ewait r_L

// statement s_i for
// right transitions
// d_i : q --a/b,R--> q'
assume state = q;
assume sym = a;
post r_R ← p b r_R d;

state := q';
// overwrites sym
ewait r_L
```

Fig. 9. The two-region task-passing program $P_{\mathcal{A}}$ simulating a given Turing machine $\mathcal{A}$.

By connecting the configurations of $\langle q, w_1, w_2 \rangle$ of $\mathcal{A}$ to the chain of tasks in region $\mathtt{r}_L$—corresponding to the cells of w_1—and the chain of tasks in region $\mathtt{r}_R$—corresponding to the cells of w_2—it is routine to show that $P_{\mathcal{A}}$ faithfully simulates precisely the runs of $\mathcal{A}$. As we assume $\mathcal{A}$ moves to the left upon encountering the accepting state q_f, we need only check reachability of the valuation $\ell_f \stackrel{\text{def}}{=} \{\mathtt{state} = q_f, \mathtt{sym} = a_0, \mathtt{done} = \mathbf{true}\}$ to know whether or not $\mathcal{A}$ has an accepting run. □

PROPOSITION 4.4. *$\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if ℓ_f is reachable in $P_{\mathcal{A}}$.*

Thus state reachability in $P_{\mathcal{A}}$ solves language emptiness for $\mathcal{A}$.

Using only a single region, it will not be possible to create two independent, unbounded task chains. However, if the program is allowed to be recursive, we can leverage the unbounded procedure stack as an additional, independent, unbounded data structure.

PROOF (b). By reduction from the language emptiness problem for Turing machines, let $\mathcal{A} = \langle Q, \Sigma, \hookrightarrow \rangle$ be a Turing machine with transitions $\hookrightarrow = \{d_1, \ldots, d_j\}$, and let $q_0, q_f \in Q$. We assume, without loss of generality, that upon entering the accepting state q_f, $\mathcal{A}$ performs a sequence of left moves until reaching the end of the tape; that is, $\langle q_f, a, \mathrm{L}, a, q_f \rangle \in \hookrightarrow$ for all $a \in \Sigma$. For any fixed $a_0 \in \Sigma$, we define a single-region task-passing program $P_{\mathcal{A}}$ in Figure 10, with an initial `main` procedure, an auxiliary procedure `p`, and a return-value handler `d`.

As in the proof of (a), our program $P_{\mathcal{A}}$ simulates the cells to the right of $\mathcal{A}$'s tape head using a linked list of procedure frames. However, as we have not here an extra region with which to encode a second linked list, we instead encode $\mathcal{A}$'s transitions *inside* the context of procedure `p`, and use the procedure frames below `p` on the procedure stack to store the cells to the left of $\mathcal{A}$'s tape head. Technically, we consider the bottom-most `p`-frame whose next statement is not **ewait** `r` to represent $\mathcal{A}$'s current tape head. For a right move, $P_{\mathcal{A}}$ simply waits for the right-neighboring cell to return; when the **ewait** `r`

```
proc main ()
    reg r
    var state, start,
        right: Q
    var sym, right_sym: Σ
    var done: 𝔹 = false
    // guess the contents
    // of a valid tape input
    while ⋆ do
        post r ← p ⋆ r d;
    ewait r;
    assume right = q0;
    // check: qf reachable?
    done := true;
    start := q0;
    sym := a0;
    right_sym := a0;
    return

proc p (reg r, var sym: Σ)
    var state, start,
        right: Q
    var right_sym: Σ
    start := ⋆;
    state := start;
    // simulate the machine
    while ⋆ do
        if ⋆ then s1
        else if ⋆ then s2
        else if ...
        else sj;
    return
// return-value handler
d(q,q',a):
    right := q;
    state := q';
    right_sym := a

// statement si for
// left transitions
// di : q --a/b,L--> q'
assume state = q;
assume sym = a;
return (start, q', b);
// a previous right-move
// replenishes this cell
// statement si for
// right transitions
// di : q --a/b,R--> q'
assume state = q;
assume sym = a;
sym := b;
ewait r;
// return from the right
assume right = q';
post r ← p right_sym r d
```

Fig. 10. The single-region task-passing program $P_{\mathcal{A}}$ simulating a given Turing machine $\mathcal{A}$.

statement progresses, this represents a left move back from the right-neighboring tape cell (which may occur much later in the simulated run) upon which we ensure the target state q' of the current cell's last move was the state where the right-neighboring cell began—this state q' is nondeterministically guessed when the right-neighboring tape cell was created. (Note that this state-guessing-and-checking gymnastics is necessary, since the **ewait** statement can only receive values from the consumed task.) Finally, to conserve $\mathcal{A}$'s tape entirely, we recreate the consumed right neighbor with the symbol its last transition wrote, and its r-pointer to its right neighbor. For a left move, we simply return the guessed initial state for the left-neighboring cell to validate, as just described, along with the target symbol and state of the given transition.

By connecting the configurations of $\langle q, w_1, w_2\rangle$ of $\mathcal{A}$ to the chain of awaiting tasks—corresponding to the cells of w_1—and the chain of posted tasks—corresponding to the cells of w_2—it is routine to show that $P_{\mathcal{A}}$ faithfully simulates precisely the runs of $\mathcal{A}$. As we assume $\mathcal{A}$ moves to the left upon encountering the accepting state q_f, we need only check reachability of the valuation

$$\ell_f \stackrel{\text{def}}{=} \{\ \texttt{state}{=}q_f,\ \texttt{start}{=}q_0,\ \texttt{right}{=}q_0,\ \texttt{sym}{=}a_0,\ \texttt{right\_sym}{=}a_0,\ \texttt{done}{=}\textbf{true}\ \}$$

to know whether or not $\mathcal{A}$ has an accepting run. □

PROPOSITION 4.5. *$\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if ℓ_f is reachable in $P_{\mathcal{A}}$.*

Thus state reachability of $P_{\mathcal{A}}$ solves language emptiness for $\mathcal{A}$.

4.2. A Decidable Case with Task Passing

On the contrary, when only one region is granted and recursion is not allowed, neither of the previous reductions work. Without recursion we can bound the procedure stack, and then we can show that single-stack machine suffices to encode the single unbounded chain of tasks, though only in the "nonaliasing" case (see Section 9.1), where each frame contains at most one pending task handle (per region). Otherwise, even with only a single region and without recursion, state reachability becomes as hard as reachability in vector addition systems when the **await** statement is allowed

(see Theorem 12.5 of Section 12.2) or coverability when **await** is disallowed (see Theorem 9.9 of Section 9.3).

THEOREM 4.6. *The state-reachability problem for nonaliasing single-region nonrecursive finite-value task-passing parallel programs is PTIME-complete for fixed task depth, and in EXPTIME-complete when task depth is not fixed.*

PROOF. Let P be a nonaliasing finite-value single-region nonrecursive task-passing parallel program with finite sets of procedures Procs, values Vals, regions Regs, and return-value handlers Rets, and let $\ell_f \in \mathsf{Vals}$ be a target reachable value. For brevity we suppose that P does not contain **await** statements, though this proof is easily extended to include them. Since P is nonrecursive, there is maximum task depth $N \in \mathbb{N}$—that is, N is the maximum length of a sequence $p_0 p_1 \ldots \in \mathsf{Procs}^*$ such that each p_i contains a post to p_{i+1}. Without loss of generality, suppose ℓ_f is reachable in P only if ℓ_f is reachable in a procedure frame where the current statement is s_f.

We construct a pushdown automaton $\mathcal{A}_P = \langle Q, \Sigma, \Gamma, \hookrightarrow \rangle$ whose states $Q \stackrel{\text{def}}{=} \{\vec{t} \in \mathsf{Tasks}^* : |\vec{t}| \leq N\}$ represent N-bounded task sequences, and whose stack symbols $\Gamma \stackrel{\text{def}}{=} \{d(v) : d \in \mathsf{Rets}, v \in \mathsf{Vals}\}$ store return-value handlers applied to return values. In this way a state $t_0 t_1 \ldots t_i \in Q$ represents a computation of P in which each t_{j-1} $(0 < j \leq i)$ is a task posted by t_j. Note that this finite representation is only possible since we know the task depth is bounded by N. Given this state representation, we define the transition relation $\hookrightarrow$ of $\mathcal{A}_P$ as follows.[3]

$$\frac{t_1 \xrightarrow[P]{\text{seq}} t_2}{t_1 \cdot \vec{t} \hookrightarrow t_2 \cdot \vec{t}} \qquad \frac{v \in e(T) \qquad t = \langle v, s_p, d \rangle}{T[\mathbf{post}\ r \leftarrow p\ e\ r\ d] \cdot \vec{t} \hookrightarrow t \cdot T[\mathbf{skip}] \cdot \vec{t}}$$

$$\frac{s \in \mathsf{rvh}(t_1)}{t_1 \cdot t_2 \cdot \vec{t} \xhookrightarrow{\text{push } s} t_2 \cdot \vec{t}} \qquad \frac{}{T[\mathbf{ewait}\ r] \cdot \vec{t} \xhookrightarrow{\text{pop } s} T[s] \cdot \vec{t}}$$

Essentially, $\mathcal{A}_P$ executes procedures immediately as they are posted, and pushes their applied return-value handlers $d(v)$ on the $\mathcal{A}_P$'s stack when returning; the handlers are popped, and immediately applied, when **ewait** statements are encountered.

Finally, given an initial condition $\iota = \langle p_0, \ell_0 \rangle$ and target value ℓ_f of P, we define an initial state $q_0 \stackrel{\text{def}}{=} \langle \ell_0, s_{p_0}, d \rangle$, and a final state $q_f \stackrel{\text{def}}{=} \langle \ell_f, s_f, d \rangle$, for an arbitrarily fixed $d \in \mathsf{Rets}$, where s_f is defined as before. □

PROPOSITION 4.7. *$\mathcal{A}_P(q_0, q_f) \neq \emptyset$ if and only if ℓ_f is reachable in P from ι.*

Though the size of $\mathcal{A}_P$ is exponential in N, $\mathcal{A}_P$ is polynomial in P for fixed N. Since language emptiness is decidable in polynomial time for pushdown automata, our procedure gives a polynomial-time algorithm for state reachability when N is fixed, though otherwise exponential in N.

PTIME-hardness when task depth is fixed follows by reduction from language emptiness for pushdown automata, using our single region to form a task chain encoding the pushdown stack. EXPTIME-hardness for variable task depth N follows by reduction from the problem of determining whether the intersection between N finite-state automata and one pushdown automaton is empty [Esparza et al. 2003]: we again encode the stack with a task chain, and use the (bounded) procedure stack to store the states of each of the N automata.

[3]We write $q_1 \xhookrightarrow{\text{push } s} q_2$, respectively, $q_1 \xhookrightarrow{\text{pop } s} q_2$, to denote the transition $q_1, s_0 \hookrightarrow q_2, s s_0$, respectively, $q_1, s \hookrightarrow q_2, \varepsilon$.

$$\textsc{Intra}\ \frac{t_1 \xrightarrow[P]{\text{seq}} t_2}{\langle t_1, m\rangle c \xrightarrow[P]{\text{rpp}} \langle t_2, m\rangle c} \qquad \textsc{Post}\ \frac{v \in e(T) \qquad m' = m \cup (r \mapsto \langle v, s_p, d\rangle)}{\langle T[\mathbf{post}\ r \leftarrow p\ e\ d], m\rangle c \xrightarrow[P]{\text{rpp}} \langle T[\mathbf{skip}], m'\rangle c}$$

$$\exists\textsc{wait}\ \frac{m = (r \mapsto t_2) \cup m'}{\langle T_1[\mathbf{ewait}\ r], m\rangle c \xrightarrow[P]{\text{rpp}} \langle t_2, \emptyset\rangle\langle T_1[\mathbf{skip}], m'\rangle c} \qquad \textsc{Return}\ \frac{s \in \mathsf{rvh}(t_1)}{\langle t_1, m_1\rangle\langle T_2[\mathbf{skip}], m_2\rangle c \xrightarrow[P]{\text{rpp}} \langle T_2[s], m_1 \cup m_2\rangle c}$$

$$\forall\textsc{wait-Done}\ \frac{m(r) = \emptyset}{\langle T[\mathbf{await}\ r], m\rangle c \xrightarrow[P]{\text{rpp}} \langle T[\mathbf{skip}], m\rangle c} \qquad \forall\textsc{wait-Next}\ \frac{m = (r \mapsto t_2) \cup m'}{\langle T_1[\mathbf{await}\ r], m\rangle c \xrightarrow[P]{\text{rpp}} \langle t_2, \emptyset\rangle\langle T_1[\mathbf{skip};\ \mathbf{await}\ r], m'\rangle c}$$

Fig. 11. The stack-based transition relation for sequentially executing recursively parallel programs without task passing.

5. PROGRAMS WITHOUT TASK PASSING

Due to the undecidability result of Theorem 4.3 and our desire to compare the analysis complexities of parallel programming models, we consider, henceforth, unless otherwise specified, only non-task-passing programs, simplifying program syntax by writing **post** $r \leftarrow p\ e\ d$. When task passing is not allowed, region valuations need not store an entire configuration for each newly posted task, since the posted task's initial region valuation is empty. As this represents a significant simplification on which our subsequent analysis results rely, we redefine here a few key notions.

A *region valuation* is a (non-nested) mapping $m : \mathsf{Regs} \to \mathbb{M}[\mathsf{Tasks}]$ from regions to multisets of tasks, a *frame* $\langle t, m\rangle$ is a task $t \in \mathsf{Tasks}$ paired with a region valuation m, and a *configuration* c is a sequence of frames representing a procedure activation stack. Figures 5 and 11 define the sequential transition relation $\to^{rpp}$ of recursively parallel programs without task passing. The definitions of *sequential execution*, *initial*, and *reachable* are nearly identical to their task-passing parallel and sequential counterparts. Since pending tasks need not store initial region valuations in non-task-passing programs, this simpler semantics is equivalent to the previous stack based semantics.

LEMMA 5.1. *For all initial conditions ι of a non-task-passing program P, the valuation ℓ is reachable in P from ι by a sequential execution with task passing if and only if ℓ is reachable in P from ι by a sequential execution without task passing.*

PROOF. The mapping between the transitions $\to^{rpp/s}$ with task passing, and those $\to^{rpp}$ without, is nearly immediate, given that $\vec{r} = \varepsilon$ in all **post** $r \leftarrow p\ e\ \vec{r}\ d$ statements of non-task-passing programs, since $m \mid \varepsilon = \emptyset$. □

Even with this simplification, we do not presently know whether the state-reachability problem for (finite-value) recursively parallel programs is decidable, in general. In the following sections, we identify several decidable, and in some cases tractable, restrictions to our program model which correspond to the concurrency mechanisms found in real-world parallel programming languages. Our decision procedures operate on a fundamental formal model based on vector addition systems with recursion, to which finite-value recursively parallel programs without task passing succinctly reduce.

6. A FUNDAMENTAL FORMAL MODEL FOR RECURSIVELY PARALLEL PROGRAMS

Recursively parallel programs without task passing bear a close resemblance to recursive sequential programs, except that each procedure frame besides carrying a local valuation carries an unbounded multiset of pending tasks' procedure frames. In this section we propose a fundamental formal program model with which to reason automatically about recursively parallel programs, analogously to the fundamental model of pushdown automata used to reason about recursive sequential programs. In what follows we introduce a model based on vector addition systems with which we will keep count of the pending tasks using unbounded integer vectors. Although we introduce zero-testing transitions to express programs' **await** transitions, this is solely for the convenience of defining the universal translation from recursively parallel programs of Section 7; our algorithms in the following sections do not reason directly about vector addition systems with zero-test edges, which is a Turing-powerful model.

6.1. Vector Addition Systems with Recursion and Zero Tests

Fix $k \in \mathbb{N}$. A *recursive vector addition system (with states) and zero tests (RVASS+Z)* $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ *of dimension* k is a finite set Q of states, along with a finite set $\hookrightarrow = \hookrightarrow_1 \uplus \hookrightarrow_2 \uplus \hookrightarrow_3$ of transitions partitioned into *additive* transitions $\hookrightarrow_1 \subset Q \times \mathbb{N}^k \times \mathbb{N}^k \times Q$, *recursive* transitions $\hookrightarrow_2 \subseteq Q \times Q \times Q \times Q$, and *zero-test* transitions $\hookrightarrow_3 \subseteq Q \times \wp(\{1, \ldots, k\}) \times Q$. We write

$$q \xhookrightarrow{\vec{n}_1 \vec{n}_2} q' \text{ when } \langle q, \vec{n}_1, \vec{n}_2, q' \rangle \in \hookrightarrow_1,$$
$$q \xhookrightarrow{q_1 q_2} q' \text{ when } \langle q, q_1, q_2, q' \rangle \in \hookrightarrow_2, \text{ and}$$
$$q \xhookrightarrow{\text{zero}(I)} q' \text{ when } \langle q, I, q' \rangle \in \hookrightarrow_3 .$$

A *recursive vector addition system (with states) (RVASS)* is an RVASS+Z which does not contain zero-test transitions, and a *(nonrecursive) vector addition system (with states) (VASS)* is an RVASS+Z which contains only additive transitions.

An *(RVASS) frame* $\langle q, \vec{n} \rangle$ is a state $q \in Q$ along with a vector $\vec{n} \in \mathbb{N}^k$, and an *(RVASS) configuration* $c \in (Q \times \mathbb{N}^k)^+$ is a nonempty sequence of frames representing a stack of nonrecursive subcomputations. The transition relation $\rightarrow^{rvas}$ for recursive vector addition systems is defined in Figure 12. The ADDITIVE rule updates the top frame $\langle q, \vec{n} \rangle$ by subtracting the vector $\vec{n}_1$ from $\vec{n}$, adding the vector $\vec{n}_2$ to the result, and updating the control state to q'. The CALL rule pushes on the frame stack a new frame $\langle q_1, \mathbf{0} \rangle$ from which the RETURN rule will eventually pop at some point when the control state is q_2; when this happens, the vector $\vec{n}_1$ of the popped frame is added to the vector $\vec{n}_2$ of the frame below. We describe an application of the CALL (respectively, RETURN) rule as a *call (respectively, return) transition*. Finally, the ZERO rule proceeds only when $\vec{n}(i) = 0$ for all $i \in I$ in the top-most frame's vector $\vec{n}$. An *execution* of an RVASS+Z $\mathcal{A}$ (from c_0 to c_j) is a configuration sequence $c_0 c_1 \ldots c_j$ where $c_i \rightarrow^{rvas} c_{i+1}$ for $0 \leq i < j$. We say the configuration $\langle q, \vec{n} \rangle$ (respectively, the state q) is *reachable* in $\mathcal{A}$ from q_0 when there exists an execution from $\langle q_0, \mathbf{0} \rangle$ to $\langle q, \vec{n} \rangle \vec{c}$ for some frame sequence $\vec{c}$ (respectively, and for some $\vec{n} \in \mathbb{N}^k$).

Problem 2 (*State Reachability*). The *state-reachability problem* is to determine whether a given state q_f of an RVASS+Z $\mathcal{A}$ is reachable from a given state q_0.

Problem 3 (*Frame Reachability*). The *(frame-)reachability problem* is to determine whether a given frame $\langle q, \vec{n} \rangle$ of an RVASS+Z $\mathcal{A}$ is reachable from a given q_0.

Note that in the literature on vector addition systems and Petri nets [Karp and Miller 1969; Lipton 1976; Mayr 1981; Rackoff 1978] what we refer to as "frame

$$\textsc{Additive}\ \frac{q \xrightarrow{\vec{n}_1\vec{n}_2} q' \qquad \vec{n} \geq \vec{n}_1}{\langle q,\vec{n}\rangle c \xrightarrow{\text{rvas}} \langle q',\vec{n} \ominus \vec{n}_1 \oplus \vec{n}_2\rangle c} \qquad \textsc{Zero}\ \frac{q \xrightarrow{\text{zero}(I)} q' \qquad \forall i \in I.\vec{n}(i) = 0}{\langle q,\vec{n}\rangle c \xrightarrow{\text{rvas}} \langle q',\vec{n}\rangle c}$$

$$\textsc{Call}\ \frac{q \xrightarrow{q_1q_2} q'}{\langle q,\vec{n}\rangle c \xrightarrow{\text{rvas}} \langle q_1,\mathbf{0}\rangle\langle q,\vec{n}\rangle c} \qquad \textsc{Return}\ \frac{q \xrightarrow{q_1q_2} q'}{\langle q_2,\vec{n}_1\rangle\langle q,\vec{n}_2\rangle c \xrightarrow{\text{rvas}} \langle q',\vec{n}_1 \oplus \vec{n}_2\rangle c}$$

Fig. 12. The transition relation for recursive vector addition systems. To simplify presentation, we assume that there is at most one recursive transition originating from each state, that is, for all $q \in Q$, $\left|\delta_2 \cap (\{q\} \times Q^3)\right| \leq 1$. We denote by $\mathbf{0}$ the vector $\langle 0,0,\ldots,0\rangle$, and by $\oplus$ and $\ominus$ the usual vector addition and subtraction operators.

reachability" is usually called "(configuration, or vector) reachability"—though there a "configuration" corresponds to a single RVASS frame—and what we refer to as "state reachability" is often called "coverability," for Petri nets and vector addition systems without states. These are classically studied problems for nonrecursive systems, with an EXPSPACE lower bound [Lipton 1976], and a matching upper bound for coverability [Rackoff 1978]. Though the general reachability problem is known to be decidable [Kosaraju 1982; Mayr 1981], thus far the only known algorithms are nonprimitive recursive. While Esparza and Nielsen [1994] survey the history of these and related problems, Leroux's [2011] recent work develops the state-of-the-art reachability algorithm.

LEMMA 6.1. [LIPTON 1976; RACKOFF 1978]. *The state-reachability problem for vector addition systems is EXPSPACE-complete.*

LEMMA 6.2. [LIPTON 1976; MAYR 1981]. *The reachability problem for vector addition systems is EXPSPACE-hard, and decidable.*

6.2. Relation to Vector Addition Systems with Branching

Recently Verma and Goubault-Larrecq [2005] have studied a branching extension to vector addition systems (BVASS), which besides additive transitions include branch-merging transitions

$$\textsc{Merge}\ \frac{q_1 \xrightarrow{q_2} q_1' \qquad \langle q_0,\mathbf{0}\rangle \xrightarrow{\text{bvas}} * \langle q_2,\vec{n}_2\rangle}{\langle q_1,\vec{n}_1\rangle \xrightarrow{\text{bvas}} \langle q_1',\vec{n}_1 \oplus \vec{n}_2\rangle},$$

which combine the vectors $\vec{n}_1$ and $\vec{n}_2$ obtained along two separate BVASS computation paths starting from fixed "axiom" states q_0. Demri et al. [2009,] proved the BVASS coverability problem is 2EXPTIME-complete.

LEMMA 6.3. [DEMRI ET AL. 2009]. *The coverability problem for branching vector addition systems is 2EXPTIME-complete.*

Our modeling of recursively parallel programs in Section 7 is naturally conceived with the RVASS formalism, since programs pass parameters to recursive procedure calls. Whereas recursive computations in RVASS begin with a CALL transition which

$$\frac{t_1 \xrightarrow[P]{\text{seq}} t_2}{t_1 \xhookrightarrow{\mathbf{00}} t_2} \qquad \frac{v_0 \in e(T) \qquad t_0 = \langle v_0, s_p, \mathtt{d}_{call}\rangle \qquad (\mathtt{l} := v_f) \in \mathsf{rvh}(t_f)}{T\big[\mathbf{call}\ \mathtt{l} := p\ e\big] \xhookrightarrow{t_0 t_f} T\big[\mathtt{l} := v_f\big]}$$

$$\frac{v_0 \in e(T) \qquad i = \mathsf{cn}(r, \langle v_0, s_p, d\rangle)}{T\big[\mathbf{post}\ r \leftarrow p\ e\ d\big] \xhookrightarrow{\mathbf{0}\vec{n}_i} T\big[\mathbf{skip}\big]} \qquad \frac{i = \mathsf{cn}(r, t_0) \qquad s \in \mathsf{rvh}(t_f)}{T\big[\mathbf{ewait}\ r\big] \xhookrightarrow{\vec{n}_i \mathbf{0}} \big\langle T\big[\mathbf{skip}\big], t_0, t_f\big\rangle \xhookrightarrow{t_0 t_f} T[s]}$$

$$\frac{i = \mathsf{cn}(r, t_0) \qquad s \in \mathsf{rvh}(t_f)}{T\big[\mathbf{await}\ r\big] \xhookrightarrow{\vec{n}_i \mathbf{0}} \big\langle T\big[\mathbf{skip}\big], t_0, t_f\big\rangle \xhookrightarrow{t_0 t_f} T\big[s;\ \mathbf{await}\ r\big]} \qquad \frac{I = \{\mathsf{cn}(r,t) : t \in \mathsf{Tasks}\}}{T\big[\mathbf{await}\ r\big] \xhookrightarrow{\mathrm{zero}(I)} T\big[\mathbf{skip}\big]}$$

Fig. 13. The transitions of the RVASS+Z $\mathcal{A}_P$ encoding the behavior of a finite-data recursively parallel program P without task passing.

passes an initial computation state, the BVASS MERGE transition only considers the state *reached* in a branch's subcomputation. Still, from the computational point of view RVASS (without zero-test transitions) and BVASS are equivalent.

LEMMA 6.4. *The state-reachability problem for RVASS, without zero-test transitions, is polynomial-time equivalent to the coverability problem for BVASS.*

PROOF. We demonstrate this equivalence by a two-way translation between BVASS and RVASS. Given a BVASS $\mathcal{A}$ with axiom state q_0, we construct an RVASS $\mathcal{A}'$ with the same states and additive transitions as $\mathcal{A}$, which has additionally a recursive transition $q_1 \xrightarrow{q_0 q_2} q_1'$ for each branch-merging transition $q_1 \xrightarrow{q_2} q_1'$ of $\mathcal{A}$. It is easily seen that $\mathcal{A}$ and $\mathcal{A}'$ have the same set of reachable states.

In the other direction, given an RVASS $\mathcal{A}$ with states Q we construct a BVASS $\mathcal{A}'$ with states $Q' = Q^2 \cup \{q_\star\}$ including an axiom state $q_\star$. The transitions of $\mathcal{A}'$ include:

— an additive transition $q_\star \xhookrightarrow{\mathbf{00}} \langle q_0, q_0\rangle$ for each state $q_0 \in Q$,
— an additive transition $\langle q_0, q\rangle \xhookrightarrow{\vec{n}_1 \vec{n}_2} \langle q_0, q'\rangle$ for each $q \xhookrightarrow{\vec{n}_1 \vec{n}_2} q'$ of $\mathcal{A}$ and $q_0 \in Q$, and
— a branch-merging transition $\langle q_0, q\rangle \xrightarrow{\langle q_1, q_2\rangle} \langle q_0, q'\rangle$ for each $q \xrightarrow{q_1 q_2} q'$ of $\mathcal{A}$ and $q_0 \in Q$.

Essentially $\mathcal{A}'$ simulates the recursive transitions of $\mathcal{A}$ by guessing the initial state q_0 which should have been passed to each recursive transition, carrying along q_0 through each additive transition, and later merging only those subcomputations which began from the initial state passed by the recursive transition. It is straightforward to show that a state $q \in Q$ is reachable from q_0 in $\mathcal{A}$ if and only $\langle q_0, q\rangle$ is reachable in $\mathcal{A}'$. □

COROLLARY 6.5. *The state-reachability problem for RVASS is 2EXPTIME-complete.*

7. ENCODING RECURSIVELY PARALLEL PROGRAMS WITH RVASS+Z

When the value set Vals of a given program P is taken to be finite, the set Tasks also becomes finite since there are finitely many statements and return-value handlers which occur in P. As finite-domain multisets are equivalently encoded with a finite number of counters (i.e., one counter per element), we can encode each region valuation $m \in \mathsf{Regs} \to \mathbb{M}[\mathsf{Tasks}]$ by a vector $\vec{n} \in \mathbb{N}^k$ of counters, where $k = |\mathsf{Regs} \times \mathsf{Tasks}|$. To

clarify the correspondence, we fix an enumeration $\mathsf{cn} : \mathsf{Regs} \times \mathsf{Tasks} \to \{1, \ldots, k\}$, and associate each region valuation m with a vector $\vec{n}$ such that for all $r \in \mathsf{Regs}$ and $t \in \mathsf{Tasks}$, $m(r)(t) = \vec{n}(\mathsf{cn}(r,t))$. Let $\vec{n}_i$ denote the unit vector of dimension i, that is, $\vec{n}_i(i) = 1$ and $\vec{n}_i(j) = 0$ for $j \neq i$.

Given a finite-data recursively parallel program P without task passing, we associate a corresponding recursive vector addition system $\mathcal{A}_P = \langle Q, \hookrightarrow \rangle$ with zero-test transitions. We define $Q \stackrel{\text{def}}{=} \mathsf{Tasks} \cup \mathsf{Tasks}^3$, and define $\hookrightarrow$ formally in Figure 13. Intra-procedural transitions translate directly to additive transitions. The **call** statements are handled by recursive transitions between entry and exit points t_0 and t_f of the called procedure. (We interpret here the **call** statement only for illustration and a slight optimization; technically this is not necessary, since we have already given the semantics of **call** by reduction to **post** and **ewait** in Section 3.2.) The **post** statements are handled by additive transitions that increment the counter corresponding to a region-task pair. The **ewait** statements are handled in two steps: first an additive transition decrements the counter corresponding to region-task pair $\langle r, t_0 \rangle$, then a recursive transition between entry and exit points t_0 and t_f of the corresponding procedure is made, applying the return-value handler of t_f upon the return. (Here we use an intermediate state $\langle T[\mathbf{skip}], t_0, t_f \rangle \in Q$ to connect the two transitions, in order to differentiate the intermediate steps of other **ewait** transitions.) The **await** statements are handled similarly, except the **await** statement must be repeated again upon the return. Finally, a zero-test transition allows $\mathcal{A}_P$ to step **await** statements.

LEMMA 7.1. *For all programs P without task passing, procedures $p_0 \in \mathsf{Procs}$, and values $\ell_0, \ell_f \in \mathsf{Vals}$, ℓ_f is reachable from $\langle \ell_0, p_0 \rangle$ in P if and only if there exist $s \in \mathsf{Stmts}$ and $d_0, d \in \mathsf{Rets}$ such that $\langle \ell_f, s, d \rangle$ is reachable from $\langle \ell_0, s_{p_0}, d_0 \rangle$ in $\mathcal{A}_P$.*

PROOF. Ignoring intermediate states $\langle t_1, t_2, t_3 \rangle \in Q$, the frames $\langle t, \vec{n} \rangle$ of $\mathcal{A}_P$ correspond directly to frames $\langle t, m \rangle$ of the given program P, given the aforementioned correspondence between vectors and region valuations. This correspondence between frames indeed extends to configurations, and ultimately to the state-reachability problems between $\mathcal{A}_P$ and P. □

Our analysis algorithms in the following sections use Lemma 7.1 to compute state reachability of a program P without task passing by computing state reachability on the corresponding RVASS(+Z) $\mathcal{A}_P$. Generally speaking, our algorithms compute sets of region valuation vectors

$$\mathsf{sms}(t_0, t_f, P) \stackrel{\text{def}}{=} \{\vec{n} \in \mathbb{N}^k : \langle t_0, \mathbf{0} \rangle \xrightarrow[\mathcal{A}_P]{\text{rvas}} * \langle t_f, \vec{n} \rangle\},$$

summarizing the execution of a procedure between an entry point t_0 and exit point t_f, where we write $\to^{\text{rvas}}_{\mathcal{A}_P} *$ to denote zero or more applications of $\to^{\text{rvas}}_{\mathcal{A}_P}$. Given an effective way to compute such a function, we could systematically replace inter-procedural program steps (i.e., of the **call**, **ewait**, and **await** statements) with intra-procedural edges performing their net effect. Note, however, that even if the set of tasks is finite, the set $\mathsf{sms}(t_0, t_f, \mathcal{A}_P)$ of summaries between t_0 and t_f need not be finite; the ability to compute this set is thus the key to our summarization-based algorithms in the following sections.

8. SINGLE-WAIT PROGRAMS

Definition 8.1 (*Single Wait*). A *single-wait program* is a program which does not contain the **await** statement.

Single-wait programs can wait only for a single pending task at any program point. Many parallel programming constructs can be modeled as single-wait programs.

8.1. Parallel Programming with Futures

The **future** annotation of Multilisp [Halstead 1985] has become a widely adopted parallel programming construct, included, for example, in X10 [Charles et al. 2005] and in Leijen et al.'s [2009] task parallel library. Flanagan and Felleisen [1999] provide a principled description of its semantics. The future construct leverages the procedural program structure for parallelism, essentially adding a "lazy" procedure call which immediately returns control to the caller with a placeholder for a value that may not yet have been computed, along with an operation for ensuring that a given placeholder has been filled in with a computed value. Syntactically, futures add two statements,

$$\textbf{future}\ x\ :=\ p\ e \qquad \textbf{touch}\ x,$$

where x ranges over program variables, $p \in \mathsf{Procs}$, and $e \in \mathsf{Exprs}$. Though it is not necessarily present in the syntax of a source language with futures, we assume that every read of a variable containing a value returned by a **future** statement is explicitly preceded by a **touch** statement, and that each such value is read only once. (Note that values returned by **future** statements need not necessarily be read in the procedure frames in which they are created; this allows "chaining" of value placeholders, that is, by passing a value placeholder as an argument of another **future** statement, and ultimately to undecidable program analysis problems—see Section 4.) Semantically, the **future** statement creates a new process in which to execute the given procedure, which proceeds to execute in parallel with the caller, along with all other processes created in this way. The **touch** statement on a variable x blocks execution of the current procedure until the future procedure call which assigned to x completes, returning a value with which is copied into x. Even though each procedure can only spawn a bounded number of parallel processes, that is, one per program variable, there is in general no bound on the total number of parallelly executing processes, since procedure calls, even parallel ones, are recursive.

Example 8.2. The Fibonacci function can be implemented as a recursively parallel algorithm using futures according to Figure 14. As opposed to the usual (nave) recursive sequential implementation operating in time $\mathcal{O}(2^n)$, this parallel implementation runs in time $\mathcal{O}(n)$ (assuming an unbounded number of processors).

The semantics of futures is readily expressed with task-passing programs using the **post** and **ewait** statements. Assuming a region identifier $\mathtt{r}_x$ and return handler $\mathtt{d}_x$ for each program variable x, we encode

$$\begin{array}{lll} \textbf{future}\ x := p\ e & \text{as} & \textbf{post}\ \mathtt{r}_x \leftarrow p\ e\ \vec{\mathtt{r}}\ \mathtt{d}_x \\ \textbf{touch}\ x & \text{as} & \textbf{ewait}\ \mathtt{r}_x \end{array}$$

where $\mathtt{d}_x(v) \stackrel{\text{def}}{=} x\ :=\ v$ simply assigns the return value v to the variable x, and the vector $\vec{\mathtt{r}}$ contains each $\mathtt{r}_y$ such that the variable y appears in e.

8.2. Parallel Programming with Revisions

Burckhardt et al.'s [2010] revisions model of concurrent programming proposes a mechanism analogous to a (software) version control system such as CVS and subversion, which promises to naturally and easily parallize sequential code in order to

```
proc fib (var n: ℕ)
    var x, y: ℕ
    if n < 2 then
        return 1
    else
        future x := fib (n-1);
        future y := fib (n-2);
        touch x;
        touch y;
        return x + y
```

Fig. 14. A futures-based Fibonacci function.

```
proc fib (var n: ℕ)
    revision r1, r2
    var x, y: ℕ
    if n < 2 then
        return 1
    else
        r1 := rfork (call x := fib (n-1)
            );
        r2 := rfork (call y := fib (n-2)
            );
        join r1 m_x;
        join r2 m_y;
        return x + y
```

Fig. 15. A parallel Fibonacci function with revisions.

take advantage of multiple computing cores. There, each sequentially executing process is referred to as a *revision*. A revision can branch into two revisions, each continuing to execute in parallel on its own separate copies of data, or merge a previously created revision, provided a programmer-defined *merge function* to mitigate the updates to data which each have performed. Syntactically, revisions add two statements,

$$x \;:=\; \textbf{rfork}\; s \qquad \textbf{join}\; x\; m,$$

where x ranges over program variables, and $s \in \mathsf{Stmts}$. Semantically, the **rfork** statement creates a new process to execute the given statement, which proceeds to execute in parallel with the invoker, along with all other processes created in this way. The assignment stores a *handle* to the newly created revision in a *revision variable* x, which overwrites any revision handle previously stored in x. The **join** statement on a revision variable x blocks execution of the current revision until the revision whose handle is stored in x completes; at that point the current revision's data is updated according to a programmer-supplied merge function $m : (\mathsf{Vals} \times \mathsf{Vals} \times \mathsf{Vals}) \to \mathsf{Vals}$: when v_0, v_1 are, respectively, the initial and final data values of the merged revision, and v_2 is the current data value of the current revision, the current revisions data value is updated to $m(v_0, v_1, v_2)$.

Example 8.3. The Fibonacci function can be implemented as a parallel algorithm using revisions according to Figure 15, given the programmer-supplied merge functions $m_{\mathtt{x}}$ and $m_{\mathtt{y}}$ which overwrite, respectively, the parent's values of x and y only. As opposed to the usual (naïve) recursive sequential implementation operating in time $\mathcal{O}(2^n)$, this recursively parallel implementation runs in time $\mathcal{O}(n)$ (assuming an unbounded number of processors).

The semantics of revisions is readily expressed with task-passing programs using the **post** and **ewait** statements. Assuming a region identifier $\mathtt{r}_x$ for each program variable x, and a programmer-supplied merge function m, we encode

$$\begin{array}{lll} x := \textbf{rfork}\; s & \text{as} & \textbf{post}\; \mathtt{r}_x \leftarrow p_s \;\mathtt{l}\; \vec{\mathtt{r}}\; \mathtt{d} \\ \textbf{join}\; x & \text{as} & \textbf{ewait}\; \mathtt{r}_x \end{array}$$

where p_s is a procedure declared as

proc p_s (**var** l: T)
  var $\mathtt{l}_0$:= l
  s;
  return ($\mathtt{l}_0$,l)

and $\mathtt{d}(\langle v_0, v_1\rangle) \stackrel{\text{def}}{=} \mathtt{l} := m(v_0, \mathtt{l}, v_1)$ updates the current local valuation based on the joined revision's initial and final valuations $v_0, v_1 \in \mathsf{Vals}$, and the joining revision's current local valuation stored in l. The vector $\vec{\mathtt{r}}$ contains each $\mathtt{r}_y$ for which the revision variable y is accessed in s.[4]

8.3. Programming with Asynchronous Procedures

Asynchronous programs [Ganty and Majumdar 2012; Jhala and Majumdar 2007; Sen and Viswanathan 2006] are becoming widely used to build reactive systems, such as device drivers, Web servers, and graphical user interfaces, with low-latency requirements. Essentially, a program is made up of a collection of short-lived tasks running one-by-one and accessing a global store, which post other tasks to be run at some later time. Tasks are initially posted by an initial procedure, and may also be generated by external system events. An *event loop* repeatedly chooses a pending task from its collection to execute to completion, adding the tasks it posts back to the task collection. Syntactically, asynchronous programs add two statements,

async p e **eventloop**

such that **eventloop** is invoked only once as the last statement of the initial procedure. Semantically, the **async** statement initializes a procedure call and returns control immediately, without waiting for the call to return. The **eventloop** statement repeatedly dispatches pending—that is, called but not yet returned—procedures, and executing them to completion; each procedure executes atomically making both synchronous calls, as well as an unbounded number of additional asynchronous procedure calls. The order in which procedure calls are dispatched is chosen nondeterministically.

We encode asynchronous programs as (nondeterministic) recursively parallel programs using the **post** and **ewait** statements. Assuming a single region identifier $\mathtt{r}_0$, we encode

async p e	as	**post** $\mathtt{r}_0 \leftarrow p'$ e d
eventloop	as	**while true do ewait** $\mathtt{r}_0$.

Supposing p has top-level statement s accessing a shared global variable g (besides the procedure parameter l), we declare p' as

proc p' (**var** l: T)
  var $\mathtt{g}_0$:= $\star$
  var g := $\mathtt{g}_0$
  s; **return** ($\mathtt{g}_0$,g).

Finally $\mathtt{d}(\langle v_0, v_1\rangle) \stackrel{\text{def}}{=}$ **assume** l = v_0; l := v_1 models the atomic update p performs from an initial (guessed) shared global valuation v_0. Guessing allows us to simulate

[4]Actually $\vec{\mathtt{r}}$ must in general be chosen nondeterministically, as each revision handle may be joined either by the parent revision or its branch.

the communication of a shared global state g, which is later ensured to have begun with v_0, which the previously executed asynchronous task had written.

9. SINGLE-WAIT ANALYSIS

The absence of **await** edges in a program P implies the absence of zero-test transitions in the corresponding recursive vector addition system $\mathcal{A}_P$. To compute state reachability in P via procedure summarization, we must summarize the recursive transitions of $\mathcal{A}_P$ by additive transitions (in a nonrecursive system) accounting for the leftover pending tasks returned by reach procedure. This is not trivial in general, since the space of possibly returned region valuations is infinite. In increasing difficulty, we isolate three special cases of single-wait programs, whose analysis problems are simpler than the general case. In the simplest "nonaliasing" case where the number of tasks stored in each region of a procedure frame is limited to one, the execution of **ewait** statements is deterministic. When the number of tasks stored in each region is not limited to one, nondeterminism arises from the choice of which completed task to pick at each **ewait** statement (see the ∃-WAIT rule of Figure 11). This added power makes the state-reachability problem at least as hard as state reachability in vector addition systems—that is, EXPSPACE-hard, though the precise complexity depends on the scope of pending tasks. After examining the PTIME-complete nonaliasing case, we examine two EXPSPACE-complete cases by restricting the scope of task handles, before moving to the 2EXPTIME-complete general case.

9.1. Single-Wait Analysis without Aliasing

Many parallel programming languages consume only the computations of precisely addressed tasks. In futures, for example, the touch x statement applies to the return value of a particular procedure—the last one whose future result was assigned to x. Similarly, in revisions, the **join** x statement applies to the last revision whose handle was stored in x. Indeed in the single-wait program semantics of each case, we are guaranteed that the corresponding region, $\mathtt{r}_x$, contains at most one task handle. Thus the nondeterminism arising (from choosing between tasks in a given region) in the ∃WAIT rule of Figure 11 disappears. Though both futures and revisions allow task passing, the following results apply to futures- and revisions-based programs which only pass pending tasks *from* consumed tasks.

Definition 9.1 (*Nonaliasing*). We say a region $r \in \mathsf{Regs}$ is *aliased* in a region valuation $m : \mathsf{Regs} \to \mathbb{M}[\mathsf{Tasks}]$ when $|m(r)| > 1$. We say r is *aliasing* in a program P if there exists a reachable configuration $C[\langle t, m\rangle]$ of P in which r is aliased in m. A *nonaliasing program* is a program in which no region is aliasing.

Note that the set of nonaliasing region valuations is finite when the number of program values is. The nonaliasing restriction thus allows us immediately to reduce the state-reachability problem for single-wait programs to reachability in recursive finite-data sequential programs. To compute state reachability we consider a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of finite-state systems iteratively underapproximating the recursive system $\mathcal{A}_P$ given from a single-wait program P. Initially, $\mathcal{A}_0$ has only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P. At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing an **ewait** transition

$$T\big[\mathbf{ewait}\ r\big] \xrightarrow{\vec{n}_j\vec{n}} T[s]\,,$$

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n}$ is reachable at t_f from t_0 in $\mathcal{A}_{i-1}$, that is, $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$—recall the definitions of the counter enumeration cn and summary set sms from Section 7. This $\mathcal{A}_0\mathcal{A}_1\ldots$ sequence is guaranteed to reach a fixed point $\mathcal{A}_k$, since the set of nonaliasing region valuation vectors, and thus

the number of possibly added edges, is finite. Furthermore, as each $\mathcal{A}_i$ is finite state, only finite state reachability queries are needed to determine the reachable states of $\mathcal{A}_k$, which are precisely the same reachable states of $\mathcal{A}_P$. Note that the number of region valuations grows exponentially in the number of regions.

THEOREM 9.2. *The state-reachability problem for nonaliasing single-wait finite-value programs is PTIME-complete for a fixed number of regions, and EXPTIME-complete in the number of regions.*

PROOF. Let P be a nonaliasing single-wait finite-value program with regions $\mathtt{r}_1, \ldots, \mathtt{r}_n$. We define a *sequential* finite-value program P_s by a code-to-code translation of P. We extend each procedure declaration **proc** p (**var** `l`: T) s with additional procedure-local variables `rg`, `rg'`, and `rv`,

proc p (**var** `l`: T)
  var `rg`[n]: R := [$\bot$, .., $\bot$]
  var `rg'` [n] : R
  var `rv`: T
  s

where R is a type containing $\bot$, and values of the record type

{ `prc`: Procs, `arg`: Vals, `rh`: Rets }.

Note that R is a finite type since Procs, Vals, and Rets are finite sets. We translate each statement **return** e into **return** (`rg`, e), each statement **post** $\mathtt{r}_i \leftarrow p\ e\ d$ into the assignment

`rg`[i] := { `prc` = p, `arg` = e, `rh` = d }

and each statement **ewait** $\mathtt{r}_i$ into the statement

assume `rg`[i] $\neq \bot$;
call (`rg'`,`rv`) := `rg`[i].`prc` `rg`[i].`arg`;
`l` := `rg`[i].`rh`;
`rg`[i] := $\bot$;
for `j` := 1 **to** n **do**
  if `rg'`[`j`] $\neq \bot$ **then** `rg`[`j`] := `rg'`[`j`]

where we assume each $d \in$ Rets is given by an expression in which `rv` is a free variable. Note that for local-scope programs (see Section 9.2), the `rg'` array will always be equal to [$\bot$, .., $\bot$] and can be safely omitted from the translation.

Since regions do not alias, it is not hard to show that state reachability for the resulting sequential program P_s is equivalent to state reachability for P. (Though technically we must check for reachability for a complete local valuation in P_s, including `l`, `rg`, `rg'`, and `rv`, we may assume without loss of generality reachability to certain values, by adding

if $\star$ **then**
  `rg` := $\star$; `rg'` := $\star$; `rv` := $\star$;
  assume false

between every statement of P_s. Since the **assume false** statement cannot continue execution, this extra conditional statement has no effect on program behavior, besides

making any valuation with $\mathtt{l} = \ell_f$ reachable, if there is some reachable valuation with $\mathtt{l} = \ell_f$.) □

PROPOSITION 9.3. *The value ℓ_f is reachable in P from ι if and only if ℓ_f is reachable in P_s from ι.*

The size of P_s is polynomial in P, while the number of variables in P_s increases linearly with the number of regions n. Thus our state-reachability problem is *PTIME*-complete for fixed n since the state reachability for sequential programs is [Chaudhuri 2008; Reps et al. 1995]. When the number n of regions is not fixed, this state-reachability problem becomes EXPTIME-complete, due to the logarithmic encoding of the program values into the n extra variables.

9.2. Local-Scope Single-Wait Analysis

Definition 9.4 (*Local Scope*). A *local-scope program* is a program in which tasks only return with empty region valuations; that is, for all reachable configurations $C[\langle t[\mathbf{return}\ e], m\rangle]$ we have $m = m_\emptyset$.

The local-scope restriction allows us to compute state reachability according to the procedure summarization algorithm outlined in Section 7, since the number of possible procedure summaries becomes bounded by the product of procedure argument and return values.

THEOREM 9.5. *The state-reachability problem for local-scope single-wait finite-value programs is EXPSPACE-complete.*

We show an equivalence between the state-reachability problems of local-scope single-wait recursively parallel programs and vector addition systems (VASS)—that is, we show the problems are polynomial-time reducible to each other. EXPSPACE-completeness follows from Lemma 6.1.

LEMMA 9.6. *The state-reachability problem for local-scope single-wait finite-value programs is polynomial-time reducible to the state-reachability problem for vector addition systems (VASS).*

PROOF. To solve state reachability in local-scope single-wait programs, we compute a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of nonrecursive vector addition systems iteratively underapproximating the recursive system $\mathcal{A}_P$ arising from a program P. The initial system $\mathcal{A}_0$ has only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P. At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing an **ewait** transition

$$T\big[\mathbf{ewait}\ r\big] \xrightarrow{\vec{n}_j\mathbf{0}} T[s]$$

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$—recall the definitions of the counter enumeration cn and summary set sms from Section 7; since P is local scope, every such $\vec{n}$ must equal $\mathbf{0}$. Since the number of possibly added edges is polynomial in P, the $\mathcal{A}_0\mathcal{A}_1\ldots$ sequence is guaranteed to reach in a polynomial number of steps a fixed point $\mathcal{A}_k$ whose reachable states are exactly those of $\mathcal{A}_P$. Thus by solving a polynomial-sized sequence of state-reachability queries in polynomial-sized VASSes $\mathcal{A}_0\mathcal{A}_1\ldots\mathcal{A}_k$, we compute state reachability in local-scope single-wait programs. □

LEMMA 9.7. *The state-reachability problem for vector addition systems (VASS) is polynomial-time reducible to the state-reachability problem for local-scope single-wait finite-value programs.*

PROOF. Let $k \in \mathbb{N}$, and let $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ be a k-dimension VASS, and let $q_0, q_f \in Q$. We construct a single-wait program $P_{\mathcal{A}}$ with an initial condition ι and target valuation ℓ_f such that $\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if ℓ_f is reachable in $P_{\mathcal{A}}$ from ι.

The program $P_{\mathcal{A}}$ contains only two procedures: an initial procedure `main` and a dummy procedure `p` which will be posted (respectively, awaited) for each addition (respectively, subtraction) performed in $\mathcal{A}$. Accordingly, the region set $\mathsf{Regs} = \{\mathtt{r}_1, \ldots, \mathtt{r}_k\}$ of $P_{\mathcal{A}}$ contains a region $\mathtt{r}_i$ per vector component. The program's local variable `l` is used to store the control state of $\mathcal{A}$, and we set $\mathsf{Vals} = Q$. Finally, let $\mathsf{Rets} = \{\mathtt{d}_{const}\}$, where $\mathtt{d}_{const}(v) \stackrel{\text{def}}{=} \mathtt{l}$; that is, $\mathtt{d}_{const}$ is the return-value handler which ignores the return value, keeping the local valuation intact.

We simulate the transitions of $\mathcal{A}$ by awaiting a task from each region $\mathtt{r}_i$ once per decrement to the ith vector component, and subsequently posting a task to each region $\mathtt{r}_i$ once per increment to the i^{th} vector component. Thus for each transition $d_j = q \xrightarrow{\vec{n}_1 \vec{n}_2} q'$, we define the statement s_j as

$$\begin{aligned}
&\textbf{assume } \mathtt{l} = q \\
&\underbrace{\textbf{ewait } \mathtt{r}_1 ; \ \ldots ; \textbf{ewait } \mathtt{r}_1}_{\vec{n}_1(1) \text{ times}} ; \ldots ; \underbrace{\textbf{ewait } \mathtt{r}_k ; \ \ldots ; \textbf{ewait } \mathtt{r}_k}_{\vec{n}_1(k) \text{ times}} ; \\
&\quad \underbrace{\textbf{post } \mathtt{r}_1 \leftarrow \mathtt{p} \star \mathtt{d}_{const} ; \ \ldots ; \textbf{post } \mathtt{r}_1 \leftarrow \mathtt{p} \star \mathtt{d}_{const}}_{\vec{n}_2(1) \text{ times}} ; \\
&\quad \ldots ; \\
&\quad \underbrace{\textbf{post } \mathtt{r}_k \leftarrow \mathtt{p} \star \mathtt{d}_{const} ; \ \ldots ; \textbf{post } \mathtt{r}_k \leftarrow \mathtt{p} \star \mathtt{d}_{const}}_{\vec{n}_2(k) \text{ times}} ; \\
&\mathtt{l} := q'.
\end{aligned}$$

Finally, the initial procedure is given by

```
proc main ()
    l := q_0;
    while ⋆ do
        if ⋆ then s_1
        else if ⋆ then s_2
        ...
        else if ⋆ then s_{|δ|}
    return.
```

Note the correspondence between configurations of $\mathcal{A}$ and $P_{\mathcal{A}}$. Each configuration $\langle q, \vec{n} \rangle$ of $\mathcal{A}$ maps directly to a configuration $\langle \langle q, s, \mathtt{d}_{const} \rangle, m \rangle$ of $P_{\mathcal{A}}$, where s is the loop statement of the initial procedure, and $|m(\mathtt{r}_i)| = \vec{n}(i)$. Given this correspondence, it follows easily that the state q_f is reachable in $\mathcal{A}$ from q_0 if and only if the valuation $\ell_f = q_f$ is reachable in $P_{\mathcal{A}}$ from $\iota = \langle \mathtt{p}_{main}, q_0 \rangle$. As there are $\mathcal{O}(|\mathcal{A}|)$ statements in $P_{\mathcal{A}}$ per transition of $\mathcal{A}$, the size of $P_{\mathcal{A}}$ is $\mathcal{O}(|\mathcal{A}|^2)$. □

9.3. Global-Scope Single-Wait Analysis

Another relatively simple case of interest is when pending tasks are allowed to leave the scope in which they are posted, but can only be consumed by a particular, statically declared task in an enclosing scope. This is the case, for example, in asynchronous programs [Sen and Viswanathan 2006], though here we allow for slightly more generality,

since tasks can be posted to multiple regions, and arbitrary control in the initial procedure frame is allowed.

Definition 9.8 (*Global Scope*). A *global-scope program* is a program in which the **ewait** (and **await**) statements are used only in the initial procedure frame.

The global-scope restriction allows us to exploit the fact that the set of unconsumed tasks posted along any recursive computation which cannot use the **ewait** (nor **await**) statements forms a semilinear set. This is also the fact exploited by Ganty and Majumdar [2012] in the less general formal model of asynchronous programs.

THEOREM 9.9. *The state-reachability problem for global-scope single-wait finite-value programs is EXPSPACE-complete.*

To proceed we show an equivalence between the state-reachability problems of global-scope single-wait recursively parallel programs and vector addition systems (VASS)—that is, we show the problems are polynomial-time reducible to each other. EXPSPACE-completeness follows since state reachability in VASS is known to be EXPSPACE-complete.

LEMMA 9.10. *The state-reachability problem for global-scope single-wait finite-value programs is polynomial-time reducible to the state-reachability problem for vector addition systems (VASS).*

PROOF. Since each noninitial procedure p of a global-scope program cannot consume tasks, the set of tasks posted by p and recursively called procedures along any execution from t_0 to t_f is a semilinear set, described by the Parikh-image of a context-free language. Following Ganty and Majumdar's [2012] approach, for each $t_0, t_f \in \mathsf{Tasks}$ we construct a polynomial-sized vector addition system $\mathcal{A}(t_0, t_f)$ characterizing this semilinear set of tasks (recursively) posted between t_0 and t_f. □

PROPOSITION 9.11. [GANTY AND MAJUMDAR 2012]. *For every pair $t_0, t_f \in \mathsf{Tasks}$, region valuation m, and $p \in \mathsf{Procs}$, there exists an execution of p from $\langle t_0, m_\emptyset \rangle$ to $\langle t_f, m \rangle$ if and only if there exists $\vec{n} \in \mathbb{N}^k$ such that $\vec{n} \in \mathcal{A}_P(t_0, t_f)$, and m and $\vec{n}$ represent the same Parikh-image.*

We use each $\mathcal{A}(t_0, t_f)$ as a component of a nonrecursive vector addition system $\mathcal{A}'_P$ representing execution of the initial frame. In particular, $\mathcal{A}'_P$ contains transitions to and from the component $\mathcal{A}(t_0, t_f)$ for each $t_0, t_f \in \mathsf{Tasks}$,

$$T[\mathbf{ewait}\ r] \xrightarrow{\vec{n}_j \mathbf{0}} \langle q_0, T[\mathbf{skip}] \rangle \quad \langle q_f, T[\mathbf{skip}] \rangle \xrightarrow{\mathbf{00}} T[s]\,,$$

for all $r \in \mathsf{Regs}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and q_0 and q_f are the initial and final states of $\mathcal{A}(t_0, t_f)$. We assume each $\mathcal{A}(t_0, t_f)$ has unique initial and final states, distinct from the states of other components $\mathcal{A}(t'_0, t'_f)$. In order to transition to the correct state $T[s]$ upon completion, $\mathcal{A}(t_0, t_f)$ carries an auxiliary state component $T[\mathbf{skip}]$. In this way, for each task t' posted to region r' in an execution between t_0 and t_f, the component $\mathcal{A}(t_0, t_f)$ does the incrementing of the $\mathsf{cn}(r', t')$-component of the region-valuation vector. As each of the polynomially many components $\mathcal{A}(t_0, t_f)$ are constructed in polynomial time [Ganty and Majumdar 2012], this method constructs $\mathcal{A}'_P$ in polynomial time, reducing state reachability in P to state reachability in the VASS $\mathcal{A}'_P$

LEMMA 9.12. *The state-reachability problem for vector addition systems (VASS) is polynomial-time reducible to the state-reachability problem for global scope single-wait finite-value programs.*

PROOF. As the program $P_{\mathcal{A}}$ constructed in the proof of Lemma 9.7 from a given VASS $\mathcal{A}$ only uses the **ewait** statement in the initial procedure, $P_{\mathcal{A}}$ is also global scope. □

9.4. The General Case of Single-Wait Analysis

In general, the state-reachability problem for finite-value single-wait programs is as hard as state reachability in recursive vector addition systems without zero-test edges, which is 2EXPTIME-complete, by Corollary 6.5.

THEOREM 9.13. *The state-reachability problem for single-wait finite-value programs is 2EXPTIME-complete.*

To proceed we show an equivalence between state reachability in single-wait recursively parallel programs, and in recursive vector addition systems without zero-test edges—that is, we show the problems are polynomial-time reducible to each other. 2EXPTIME-completeness follows from Corollary 6.5.

LEMMA 9.14. *The state-reachability problem for single-wait finite-value programs is polynomial-time reducible to the state-reachability problem for recursive vector addition systems.*

PROOF. The RVASS of $\mathcal{A}_P$ of the program P is given by Lemma 7.1 of Section 7; since P does not contain **await** statements, $\mathcal{A}_P$ does not contain zero-test edges. □

LEMMA 9.15. *The state-reachability problem for recursive vector addition systems is polynomial-time reducible to the state-reachability problem for single-wait finite-value programs.*

PROOF. Let $k \in \mathbb{N}$, let $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ be a k-dimensional RVASS, and let $q_0, q_f \in Q$. We construct a single-wait program $P_{\mathcal{A}}$ with initial condition ι and target valuation ℓ_f such that $\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if ℓ_f is reachable in $P_{\mathcal{A}}$ from ι.

The program $P_{\mathcal{A}}$ contains two types of procedures: a set of recursive procedures $\{p_q : q \in Q\}$ whose invocations will correspond to recursive transitions in $\mathcal{A}$, and a *dummy procedure* p_{D} which will be posted (respectively, awaited) for each addition (respectively, subtraction) performed in $\mathcal{A}$. Accordingly, the region set $\mathsf{Regs} = \{\mathtt{r}_1, \ldots, \mathtt{r}_k, \mathtt{r}_{call}\}$ of P contains a region $\mathtt{r}_i$ per vector component, and a *call region* $\mathtt{r}_{call}$. As the program's local variable $\mathtt{l}$ is used to store the control state of $\mathcal{A}$, we set $\mathsf{Vals} = Q$. Finally, let $\mathsf{Rets} = \{\mathtt{d}_{const}\}$, where $\mathtt{d}_{const}(v) \stackrel{\text{def}}{=} \ell$; that is, $\mathtt{d}_{const}$ is the return-value handler which ignores the return value, keeping the local valuation intact.

The top-level statement for the dummy procedure p_0 is simply **return** $\star$; the top-level statement for the other procedures p_q for $q \in Q$ will simulate all transitions of $\mathcal{A}$ and return only when the control state reaches q. Let $\hookrightarrow = \{d_1, \ldots, d_n\}$. We define s_i for each $d_i \in \hookrightarrow$ as follows. We simulate recursive transitions by calling a procedure which may only return upon reaching q_2. For each transition $d_i = q \xhookrightarrow{q_1 q_2} q'$, s_i is given by

assume $\mathtt{l} = q$;
call $\mathtt{l} := p_{q_2}\, q_1$;
$\mathtt{l} := q'$.

We simulate the additive transitions by awaiting a task from each region $\mathtt{r}_i$ once per decrement to the ith vector component, and subsequently posting a task to each region

$\mathtt{r}_i$ once per increment to the ith vector component. For each transition $d_i = q \xrightarrow{\vec{n}_1\vec{n}_2} q'$, s_i is given by

$$\begin{aligned}
&\textbf{assume } \mathtt{l} = q\\
&\underbrace{\textbf{ewait } \mathtt{r}_1 ; \ \ldots ; \textbf{ewait } \mathtt{r}_1}_{\vec{n}_1(1) \text{ times}} ; \ldots ; \underbrace{\textbf{ewait } \mathtt{r}_k ; \ \ldots ; \textbf{ewait } \mathtt{r}_k}_{\vec{n}_1(k) \text{ times}} ;\\
&\quad \underbrace{\textbf{post } \mathtt{r}_1 \leftarrow p_0 \star \mathtt{d}_{const} ; \ \ldots ; \textbf{post } \mathtt{r}_1 \leftarrow p_0 \star \mathtt{d}_{const}}_{\vec{n}_2(1) \text{ times}} ;\\
&\quad \ldots ;\\
&\quad \underbrace{\textbf{post } \mathtt{r}_k \leftarrow p_0 \star \mathtt{d}_{const} ; \ \ldots ; \textbf{post } \mathtt{r}_k \leftarrow p_0 \star \mathtt{d}_{const}}_{\vec{n}_2(k) \text{ times}} ;\\
&\mathtt{l} := q'.
\end{aligned}$$

Finally, the top-level statement for procedure p_q is

while $\star$ **do**
  if $\mathtt{l} = q$ **and** $\star$ **then return** $\star$
  else if $\star$ **then** s_1
  else if $\star$ **then** s_2
  ...
  else if $\star$ **then** s_n
  else skip.

Note the correspondence between configurations of $\mathcal{A}$ and $P_{\mathcal{A}}$. Each frame $\langle q, \vec{n} \rangle$ of $\mathcal{A}$ maps directly to a frame $\langle \langle q, s, \mathtt{d}_{const} \rangle, m \rangle$ of $P_{\mathcal{A}}$, where s is the top-level statement of some procedure $p_{q'}$, and $|m(\mathtt{r}_i)| = \vec{n}(i)$ for all $i \in \{1, \ldots, k\}$; this correspondence extends directly to the configurations of $\mathcal{A}$ and $P_{\mathcal{A}}$. It follows that the state q_f is reachable in $\mathcal{A}$ if and only if the valuation q_f is reachable in $P_{\mathcal{A}}$. As there are $\mathcal{O}(|Q|)$ statements in $P_{\mathcal{A}}$ per transition of $\mathcal{A}$, the size of $P_{\mathcal{A}}$ is $\mathcal{O}(|\mathcal{A}|^2)$. □

10. SUMMARIZATION-BASED STATE REACHABILITY FOR SINGLE-WAIT PROGRAMS

Our result of 2EXPTIME membership for state reachability in single-wait finite-value recursively parallel programs, given by Lemma 9.14, relies on Demri et al.'s [2009] proof of 2EXPTIME membership for coverability in branching vector addition systems. While asymptotically optimal, their algorithm is given by applying Savitch's theorem [Savitch 1970] to a nondeterministic algorithm which arbitrarily guesses an execution witnessing coverability, bounded by a double exponential in the size of the given BVASS, and checks witness validity in time linear in the size of the execution. Here we propose to sidestep their construction by proposing a procedure-summarization-based algorithm. Although we do not know whether our algorithm is asymptotically optimal, we suppose that such an algorithm can be more efficient in practice, since procedure summaries can be computed on-demand, as they are required for other reachability queries. The success of such a heuristic supposes that in practically occurring programs, only a fraction of the possible summaries will ever need to be computed.

To compute state reachability we consider again a sequence $\mathcal{A}_0\mathcal{A}_1 \ldots$ of nonrecursive vector addition systems successively underapproximating the recursive system $\mathcal{A}_P$ of a single-wait program P. Initially $\mathcal{A}_0$ has only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P, and at each step $i > 0$, we add to $\mathcal{A}_i$

additive edges $T[\mathbf{ewait}\ r] \xrightarrow{\vec{n}_j\vec{n}} T[s]$ summarizing ewait transitions, for some $t_0, t_f \in$ Tasks such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$. Even though the set of possible added additive edges summarizing recursive transitions is infinite, with careful analysis we construct a very simple terminating algorithm, provided we can bound the edge labels $\vec{n}$ needed to compute state reachability in $\mathcal{A}_P$. It turns out we can bound these edge labels, by realizing that the minimal vectors required to reach a target state from any given program location are bounded.

We adopt an approach based on iteratively applying backward reachability analyses in order to determine for each state t of $\mathcal{A}_P$, the set of vectors $\eta(t)$ needed to reach the target state in $\mathcal{A}_P$; we call such a function η a *coverability map*. Let us first recall some useful basic facts. Vector addition systems are monotonic with respect to the natural ordering on vectors of positive integers, that is, if a transition is possible from a frame $\langle t, \vec{n}\rangle$, it is also possible from any frame $\langle t, \vec{n}'\rangle$ such that $\vec{n}' > \vec{n}$. The ordering on vectors of integers is a Well Quasi-Ordering (WQO), that is, in every sequence of vectors $\vec{n}_0\vec{n}_1 \ldots$, there are two indices $i < j$ such that $\vec{n}_i$ is less or equal than $\vec{n}_j$. Thus, every infinite set of vectors has a finite number of minimals. A set of vectors is upward closed if whenever it contains $\vec{n}$ it also contains all vectors greater than $\vec{n}$, and such a set can be characterized by its minimals. Moreover, the set of all predecessors in a vector addition system of an upward closed set of vectors is also upward closed; therefore backward reachability analysis in these systems always terminates starting from upward closed sets [Abdulla et al. 1996; Finkel and Schnoebelen 2001].

We observe that for every task t, the set $\eta(t)$ is upward closed (by monotonicity), and therefore we need only determine its minimals. However, since our model includes recursion, we must solve several state-reachability queries on a sequence of vector addition systems with increasingly more transitions, which necessarily stabilizes. We elaborate next.

Given the recursive system $\mathcal{A}_P$ obtained from a program P, we define $\mathcal{A}_P^{\mathrm{NR}}$ as the nonrecursive system obtained by omitting the translation of **call**, **ewait**, and **await** transitions of P. In order to reason backward about inter-procedural executions to the target state, we augment our nonrecursive systems with transitions corresponding to unmatched procedure returns. The *return completion* of a VASS $\mathcal{A}$ derived from P, denoted $\mathcal{A}^{\mathrm{RC}}$, adds to $\mathcal{A}$ a transition $t_f \xrightarrow{\mathbf{00}} T[s]$ whenever t_f and $T[\mathbf{ewait}\ r]$ are, respectively, exit and return points of the same procedure, and $s \in \mathsf{rvh}(t_f)$. These extra transitions simulate procedure returns which transfer all pending tasks of the returning procedure frame without any contribution from $T[s]$'s call-site predecessor $T[\mathbf{ewait}\ r]$.

To compute the set of backward-reachable frames, we assume a subroutine BwReach which when given a VASS $\mathcal{A}$ and target state t, computes a coverability map η : Tasks $\to \wp(\mathbb{N}^k)$, mapping each state t_0 to the (possibly empty, upward closed) set of vectors $\eta(t_0)$ such that t is reachable in $\mathcal{A}$ from any $\langle t_0, \vec{n}\rangle$ such that $\vec{n} \in \eta(t_0)$. Thus every frame $\langle t_0, \vec{n}\rangle : \vec{n} \in \eta(t_0)$ is guaranteed to reach t in $\mathcal{A}$.

For a given pair t_0, t_f of entry and exit points to a procedure p, and a given coverability map η of a VASS $\mathcal{A}$ derived from P, we say the vector $\vec{n} \in \mathbb{N}^k$ is *profitable*, and write Profitable$(t_0, t_f, \eta, \vec{n})$, when for some call site $T[\mathbf{ewait}\ r]$ of p, and $s \in \mathsf{rvh}(t_f)$, $\eta(T[s]) \ominus \vec{n} \oplus \vec{n}_j \not\subseteq \eta(T[\mathbf{ewait}\ r])$, where $j = \mathsf{cn}(r, t_0)$. Intuitively, $\vec{n}$ is a profitable summary of p executing between t_0 and t_f with respect to η when adding $\vec{n}$ as a summary of p would increase the set of backward-reachable frames in $\mathcal{A}$. We then define Summarize$(t_0, t_f, \vec{n})$ as the set of summary edges $\{T[\mathbf{ewait}\ r] \xrightarrow{\vec{n}_j\vec{n}} T[s] : j = \mathsf{cn}(r, t_0), s \in \mathsf{rvh}(t_f)\}$.

Algorithm 1 computes a sequence $\mathcal{A}_0\mathcal{A}_1 \ldots$ of vector addition systems iteratively improving an underapproximation of state reachability in $\mathcal{A}_P$ by computing a coverability map η_i for each $\mathcal{A}_i$. Initially $\mathcal{A}_0$ contains only the intra-procedural transitions of $\mathcal{A}_P$,

Algorithm 1. A decision procedure for state reachability in single-wait programs. For simplicity we compute reachability to target valuations of the initial procedure frame.

Data: An initial condition ι and target valuation ℓ of a single-wait program P
Result: Whether ℓ is reachable from ι in P

1 **let** $\mathcal{A}_P$ be the RVASS obtained from P;
2 **let** t_ι, t_ℓ be corresponding initial and target states of $\mathcal{A}_P$;
3 initialize $\mathcal{A}_0$ to $\mathcal{A}_P^{\mathrm{NR}}$;
4 initialize i to 0;
5 **while** t_ℓ *is not reachable from* t_ι *in* $\mathcal{A}_i$ **do**
6 compute $\eta_i := \mathsf{BwReach}(\mathcal{A}_i^{\mathrm{RC}}, t_\ell)$;
7 **if** *there exist no* t_0, t_f *and* $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_i)$ *such that* $\mathsf{Profitable}(t_0, t_f, \eta_i, \vec{n})$ **then**
8 **return** ℓ *is not reachable from* ι *in* P;
9 **end**
10 initialize $\mathcal{A}_{i+1}$ to $\mathcal{A}_i$;
11 **foreach** t_0, t_f *and maximal* $\vec{n}$ *of* $\{\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_i) : \mathsf{Profitable}(t_0, t_f, \eta_i, \vec{n})\}$ **do**
12 add the edges $\mathsf{Summarize}(t_0, t_f, \vec{n})$ to $\mathcal{A}_{i+1}$;
13 **end**
14 increment i;
15 **end**
16 **return** ℓ *is reachable from* ι *in* P;

and in each iteration, $\mathcal{A}_{i+1}$ is constructed by adding to $\mathcal{A}_i$ the summaries of all vectors deemed profitable in $\mathcal{A}_i$. Since each $\mathcal{A}_{i+1}$ contains at least the transitions of $\mathcal{A}_i$, the η_i-sequence is nondecreasing (with respect to set inclusion), and more and more frames of each $\mathcal{A}_{i+1}$ can reach the target state; that is, for all $t \in \mathsf{Tasks}$ we have $\eta_i(t) \subseteq \eta_{i+1}(t)$. Since there can be no ever-increasing sequence of upward-closed sets of vectors over natural numbers (by the fact that the ordering on vectors of natural numbers is a WQO), the η_i sequence stabilizes after a finite number of steps, to the point where no additional vectors can be deemed profitable.

THEOREM 10.1. *Algorithm 1 decides state reachability for finite-value single-wait programs, for target valuations of the initial procedure frame.*

PROOF. Since the sequence $\eta_0\eta_1\ldots$ of coverability maps computed by Algorithm 1 is strictly decreasing, the condition of line 7 may only be false for a finite number of loop iterations. Additionally, as procedure summary edges are only added when they are realized by sequences of nonrecursive executions of $\mathcal{A}_P$, any state reachable in any $\mathcal{A}_i$ is also reachable in $\mathcal{A}_P$.

To show completeness we argue inductively over the number of CALL transitions in (backward) executions of $\mathcal{A}_P$. Clearly $\mathcal{A}_0^{\mathrm{RC}}$ captures exactly the backward executions of $\mathcal{A}_P$ without CALL transitions. Inductively, for $k \in \mathbb{N}$, any backward execution of $\mathcal{A}_P$ using $k+1$ CALL transitions must be captured by $\mathcal{A}_{k+1}^{\mathrm{RC}}$, since $\mathcal{A}_k^{\mathrm{RC}}$ captures all backward executions of $\mathcal{A}_P$ with up to k CALL transitions, including the backward execution prefix of $\mathcal{A}_P$ to the procedure entry point $\langle t_0, \mathbf{0}\rangle$ just before its last (in backwards order) CALL transition; lines 11–13 of Algorithm 1 ensure that a summarization of this last CALL transition is added in $\mathcal{A}_{k+1}$. □

PROPOSITION 10.2. *For $k \in \mathbb{N}$, t_ℓ is reachable from $\langle t, \vec{n}\rangle$ in $\mathcal{A}_P$ using k CALL transitions only if t_ℓ is reachable from $\langle t, \vec{n}\rangle$ in $\mathcal{A}_k$.*

Thus Algorithm 1 is a decision procedure for state reachability.

Remark 10.3. For simplicity we have considered state reachability to a target state in the initial procedure frame. In order to account for states reachable by executions with unmatched procedure calls, it is possible to extend Algorithm 1 to compute, using the system $\mathcal{A}_k$ computed during the last loop iteration, a sequence of nonrecursive systems $\mathcal{A}'_0\mathcal{A}'_1\ldots$ which add not procedure summary edges, but unmatched procedure *call* edges. Similarly to computing summaries in Algorithm 1, in each step i we would add transitions $T[\mathbf{ewait}\ r] \xrightarrow{\vec{n}_j\mathbf{0}} t_0 : j = \mathsf{cn}(r, t_0)$, to $\mathcal{A}'_i$ whenever $\langle t_0, \mathbf{0}\rangle$ is backward reachable from the target in $\mathcal{A}'_{i-1}$. Since the number of possibly added edges is finite, this extension also terminates, and yields a complete algorithm for state reachability to target states in *any* procedure frame.

11. MULTI-WAIT PROGRAMS

Though single-wait programs capture many parallel programming constructs, they can not express waiting for each and every of an unbounded number of tasks to complete. Some programming languages require this dual notion, expressed here with **await**.

Definition 11.1 (*Multi-wait*). A *multi-wait program* is a program which does not contain the **ewait** statement.

Thus, multi-wait programs can wait only on every pending task (in a given region) at any program point. Many parallel programming constructs can be modeled as multi-wait programs.

11.1. Parallel Programming in Cilk

The Cilk parallel programming language [Randall 1998] is an industrial-strength language with an accompanying runtime system which is used in a spectrum of environments, from modest multicore computations to massively parallel computations with supercomputers. Similarly to futures (see Section 8.1), Cilk adds a form of procedure call which immediately returns control to the caller. Instead of an operation to synchronize with a *particular* previously called procedure, Cilk only provides an operation to synchronize with *every* previously called procedure. At such a point, the previously called procedures communicate their results back to the caller one-by-one with atomically executing procedure in-lined in scope of the caller. Syntactically, Cilk adds two statements

$$\mathbf{spawn}\ p\ e\ p' \qquad \mathbf{sync},$$

where p ranges over procedures, e over expressions, and p' over procedures declared by

$$\mathbf{inlet}\ p'\ (\mathbf{var}\ \texttt{rv}\colon T)\ s.$$

Here s ranges over intra-procedural program statements containing two variables: `rv`, corresponding to the value returned from a spawned procedure, and `l`, corresponding to the local variable of the spawning procedure. Semantically, the **spawn** statement creates a new process in which to execute the given procedure, which proceeds to execute in parallel with the caller—and all other processes created in this way. The **sync** statement blocks execution of the current procedure until each spawned procedure completes, and executes its associated inlet. The inlets of each procedure execute atomically. Each procedure can spawn an unbounded number of parallel processes, and the order in which the inlets of procedures execute is chosen nondeterministically. Cilk procedures only return after all spawned processes have completed; there is thus an implicit **sync** statement before each return.

```
proc fib (var n: ℕ)
    var sum: ℕ
    if n < 2 then return 1
    else
        spawn fib (n-1) summer;
        spawn fib (n-2) summer;
        sync; // also implicit
        return sum

inlet summer (var i: ℕ)
    sum := sum + i
```

Fig. 16. A Cilk-like parallel Fibonnaci function.

```
proc fib (var n: ℕ)
    var x, y: ℕ
    if n < 2 then
        return 1
    else
        finish
          async
            call x := fib (n-1);
          async
            call y := fib (n-2);
        return x + y
```

Fig. 17. An X10-like parallel Fibonnaci function.

Example 11.2. The Fibonacci function can be written in Cilk following the recursively parallel program of Figure 16. As opposed to the usual (naïve) recursive sequential implementation operating in time $\mathcal{O}(2^n)$, this recursively parallel implementation runs in time $\mathcal{O}(n)$ (assuming an unbounded number of processors).

The semantics of Cilk is ready expressed with recursively parallel programs using the **post** and **await** statements. Assuming a region identifier $\mathtt{r}_0$, we encode

$$\begin{array}{lll} \textbf{spawn}\ p\ e\ p' & \text{as} & \textbf{post}\ \mathtt{r}_0 \leftarrow p\ e\ \mathtt{d}_{p'} \\ \textbf{sync} & \text{as} & \textbf{await}\ \mathtt{r}_0 \end{array}$$

where $\mathtt{d}_{p'}(v) \stackrel{\text{def}}{=} s_{p'}[v/\mathtt{rv}]$ executes the top-level statement of the inlet p' with input parameter v. Note that generally speaking Cilk procedures executing in parallel may contain interfering memory accesses; our model captures only noninterfering tasks, by assuming either that they access disjoint regions of memory or that branches are properly synchronized to ensure atomicity.

11.2. Parallel Programming with Asynchronous Statements

The async/finish pair of constructs in X10 [Charles et al. 2005] introduces parallelism through asynchronously executing statements and synchronization blocks. Essentially, an asynchronous statement immediately passes control to a following statement, executing itself in parallel. A synchronization block executes as any other program block, but does not pass control to the following statements/block until every asynchronous statement within has completed. Syntactically, this mechanism is expressed with two statements,

$$\textbf{async}\ s \qquad \textbf{finish}\ s$$

where s ranges over program statements. Semantically, the **async** statement creates a new process to execute the given statement, which proceeds to execute in parallel with the invoker—and all other processes created in this way. The **finish** statement executes the given statement s, then blocks execution until every process created within s has completed.

Example 11.3. The Fibonacci function can be implemented as a parallel algorithm using asynchronous statements following the program of Figure 17. As opposed to the usual (naïve) recursive sequential implementation operating in time $\mathcal{O}(2^n)$, this recursively parallel implementation runs in time $\mathcal{O}(n)$ (assuming an unbounded number of processors).

Asynchronous statements are readily expressed with (nondeterministic) recursively parallel programs using the **post** and **await** statements. Let N be the maximum depth of nested **finish** statements. Assuming region identifiers $\mathtt{r}_1, \ldots, \mathtt{r}_N$, we encode

$$\begin{array}{lll} \textbf{async}\ s & \text{as} & \textbf{post}\ \mathtt{r}_i \leftarrow p_s \star \mathtt{d} \\ \textbf{finish}\ s & \text{as} & \textbf{await}\ \mathtt{r}_i \end{array}$$

where $i - 1$ is number of enclosing **finish** statements, and p_s is a procedure declared as

proc p_s (**var** l: T)
  var $\mathtt{l}_0$:= l
  s;
  return ($\mathtt{l}_0$,l)

and $\mathtt{d}(\langle v_0, v_1\rangle) \stackrel{\text{def}}{=}$ **assume** l = v_0; l := v_1 models the update p performs from an initial (guessed) local valuation v_0. Using the same trick we have used to model asynchronous programs in Section 8.3, we model the sequencing of asynchronous tasks by initially guessing the value v_0 which the previously executed asynchronous tasks had written, and validating that value when the return-value handler of a given task is finally run. Note that although X10 allows, in general, asynchronous tasks to interleave their memory accesses, our model captures only noninterfering tasks, by assuming either data parallelism (i.e., disjoint accesses to data) or by assuming tasks are properly synchronized to ensure atomicity.

11.3. Structured Parallel Programming

So-called structured parallel constructs are becoming a standard parallel programming feature, adopted, for instance, in X10 [Charles et al. 2005] and in Leijen et al.'s [2009] task parallel library. These constructs leverage familiar sequential control structures to express parallelism. A typical syntactic instance of this is the parallel for-each loop

foreach x **in** e **do** s

where x ranges over program variables, e over expressions, and s over statements. Semantically, the **for-each** statement creates a collection of new processes in which to execute the given statement—one for-each valuation of the loop variable. After creating these processes, the **for-each** statement then blocks execution, waiting for each to complete.

The semantics of the for-each loop is readily expressed with recursively parallel programs using the **post** and **await** statements. With a region identifier $\mathtt{r}_0$, and supposing x and l are free in s,

we encode foreach x **in** e **do** s **using a series of post**s

for x **in** e **do**
  post $\mathtt{r}_0 \leftarrow p_s$ (x,$\star$) d;
await $\mathtt{r}_0$

and statement s **as a procedure**

proc p_s (**var** x: T, l: T)
  var $\mathtt{l}_0$:= l
  s;
return ($\mathtt{l}_0$,l)

where the return-value handler $\mathtt{d}(\langle v_0, v_1\rangle) \stackrel{\text{def}}{=}$ **assume** l = v_0; l := v_1 models the update p performs from an initial (guessed) local valuation v_0. Note that generally

speaking the parallel branches of for-each loops, for example, in X10, may contain interfering memory accesses; our model captures only noninterfering tasks, by assuming either that branches access disjoint regions of memory or that branches are properly synchronized to ensure atomicity.

12. MULTI-WAIT ANALYSIS

The presence of **await** edges implies the presence of zero-test transitions in the recursive vector addition system $\mathcal{A}_P$ corresponding to a multi-wait program P. As we have done for single-wait programs, we first examine the easier subcase of local-scope programs, which in the multi-wait setting corresponds to concurrency in the Cilk [Randall 1998] language[5]—recall that Cilk procedures contain implicit **sync** statements before each return—as well as structured parallel programming constructs such as the **for-each** parallel loop in X10 [Charles et al. 2005] and in Leijen et al.'s [2009] task parallel library (see Section 11.3). The concurrent behavior of the asynchronous statements (Section 11.2) in X10 [Charles et al. 2005] does not satisfy the local-scope restriction, since **async** statements can include recursive procedure calls which are nested without interpolating **finish** statements. We show, in Section 12.2, that computing state reachability without the local-scope assumption is equivalent to determining whether a particular vector is reachable in a nonrecursive vector addition system—a decidable problem which is known to be EXPSPACE-hard, but for which the only known algorithms are nonprimitive recursive. Since all multi-wait parallel languages we have encountered use only a single region, we restrict our attention here to single-region multi-wait programs.

12.1. Local-Scope Single-Region Multi-Wait Analysis

With the local-scoping restriction, executions of each procedure $p \in \mathsf{Procs}$ between entry point $t_0 \in \mathsf{Tasks}$ and exit point $t_f \in \mathsf{Tasks}$ are completely summarized by a Boolean value indicating whether or not t_f is reachable from t_0. However, as executions of p may encounter **await** statements, modeled by zero-test edges in the recursive vector addition system $\mathcal{A}_P$, computing this Boolean value requires determining the reachable program valuations between each pair of consecutive "synchronization points", that is, occurrences of the **await** statement which in principle requires deciding whether the vector $\mathbf{0}$ is reachable in a vector addition system describing execution from the program point just after the first **await** statement to the point just after the second; in other words, when $T_1\left[\textbf{await } r\right]$ and $T_2\left[\textbf{await } r\right]$ are consecutively occurring synchronization points, we must determine whether $\langle T_1[\textbf{skip}], \mathbf{0}\rangle$ can reach $\langle T_2[\textbf{skip}], \mathbf{0}\rangle$.

A careful analysis of this reachability problem reveals it does not have the EXPSPACE-hard complexity of determining vector reachability in general, due to the special structure of the reachability query. Between two consecutive synchronization points t_1 and t_2 of p, execution proceeds in two phases. In the first phase, **post** statements made by p only increment the vector valuations. In the second phase, starting when the second **await** statement is encountered, the **await** statement repeatedly consumes tasks, only decrementing the vector valuations—the vector valuations cannot be reincremented because of the local-scope restriction: each consumed task is forbidden from returning addition tasks. Due to this special structure, deciding reachability between t_1 and t_2 reduces to deciding if a particular integer linear program $\Phi(t_1, t_2)$

[5]Modulo task interleaving, which is in certain cases possible in Cilk.

has a solution. We exploit this connection to linear programming to derive an asymptotically optimal state-reachability algorithm.

THEOREM 12.1. *The state-reachability problem for local-scope multi-wait single-region finite-value programs is NP-complete.*

We show membership in NP in Lemma 12.2 by a procedure which solves a polynomial number of polynomial-sized integer linear programs, and NP-hardness in Lemma 12.4 by a reduction from circuit satisfiability [Papadimitriou 1993].

LEMMA 12.2. *The state-reachability problem for local-scope multi-wait single-region finite-value programs P over values* Vals *and return-value handlers* Rets *is reducible to solving a polynomial-length series of polynomial-sized integer linear programs.*

PROOF. Since consuming tasks in the **await** loop requires using the summaries computed for other procedures, we consider a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of nonrecursive vector addition systems iteratively underapproximating the recursive system $\mathcal{A}_P$. Initially $\mathcal{A}_0$ has only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P. At each step $i > 0$, we add to $\mathcal{A}_i$ one of two edges types. One type is an additive procedure-summary edge, used to describe a single task consumption step of an **await** transition,

$$T\big[\mathbf{await}\ r\big] \xrightarrow{\vec{n}_j\mathbf{0}} T\big[s;\ \mathbf{await}\ r\big],$$

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1}) \neq \emptyset$—recall the definitions of the counter enumeration cn and summary set sms from Section 7. The second possibility is an additive synchronization-point summary edge, summarizing an entire of sequence of program transitions between two synchronization points,

$$T_1\big[\mathbf{skip}\big] \xrightarrow{\mathbf{00}} T_2\big[\mathbf{skip}\big],$$

where $T_1\big[\mathbf{await}\ r\big], T_2\big[\mathbf{await}\ r\big] \in \mathsf{Tasks}$ are consecutive synchronization points occurring in P, and $\mathbf{0} \in \mathsf{sms}\big(T_1\big[\mathbf{skip}\big], T_2\big[\mathbf{skip}\big], \mathcal{A}_{i-1}\big)$. The procedure-summary edges are computed using only finite-state reachability between program states, using the synchronization-point summary edges, while the synchronization-point summary edges are computed by reduction to integer linear programming. As the number of possible edges is bounded polynomially in the program size, the $\mathcal{A}_0\mathcal{A}_1\ldots$ sequence is guaranteed to reach a fixed point $\mathcal{A}_k$ in a polynomial number of steps, though each step may take nondeterministic polynomial time, in the worst case, to compute solutions to integer linear programs. Given the ability to compute synchronization-point summary edges, the reachable states of $\mathcal{A}_k$ are precisely the same reachable states of $\mathcal{A}_P$.

For a given synchronization point reachability query of $\mathcal{A}_i$ from t_0 to t_f, we construct an integer linear program $\Phi_i(t_0, t_f)$. For simplicity and without loss of generality, we suppose each expression e used in **post** $r \leftarrow p\ e\ d$ statements evaluates to a singleton, that is, there exists $v' \in \mathsf{Vals}$ such that $e(v) = \{v'\}$, for all $v \in \mathsf{Vals}$, and similarly that all return-value handlers d evaluate to singletons, that is, there exists $s \in \mathsf{Stmts}$ such that $d(v) = \{s\}$, for all $v \in \mathsf{Vals}$. We fix an enumeration $\delta = \{d_1, d_2, \ldots\}$ of $\mathcal{A}_i$'s transitions, along with an enumeration $X = \{x_1, x_2, \ldots\}$ of the procedure summaries $\{\langle t_0, t_f\rangle : \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1}) \neq \emptyset\}$ computed from $\mathcal{A}_{i-1}$. Then $\Phi_i(t_0, t_f)$ contains the following variables

$$\mathrm{d}_j \text{ for each transition } d_j \in \delta\ , \text{ and}$$
$$\mathrm{x}_j \text{ for each procedure summary } x_j \in X\ ,$$

which represent, respectively, the count of each transition taken, and the count of each summary taken, in an execution of $\mathcal{A}_i$ from t_0 to t_f. For each $t \in \mathsf{Tasks}$, the program $\Phi_i(t_0, t_f)$ contains the constraint

$$\sum_{d_j=\langle \_,t\rangle} \mathrm{d}_j - \sum_{d_j=\langle t,\_\rangle} \mathrm{d}_j = \begin{cases} -1 & \text{if } t = t_0 \\ 1 & \text{if } t = t_f \\ 0 & \text{otherwise,} \end{cases}$$

to ensure that the number of $\mathcal{A}_i$-transitions taken to state t is equal to the number of $\mathcal{A}_i$-transitions taken from state t (except for the initial and final states t_0 and t_f, which are, respectively, consumed/targeted on more time than they are targeted/consumed). When $d_j \in \delta$ is an **await** transition from $T[\mathbf{await}\, r]$ to $T[s;\ \mathbf{await}\, r]$, for some task context T, which consumes a pending task whose return-value handling statement is s, we write $d_j : s$. For each possible return-value handler statement $s \in \{d(v) : d \in \mathsf{Rets}, v \in \mathsf{Vals}\}$, $\Phi_i(t_0, t_f)$ contains the constraint

$$\sum_{d_j:s} \mathrm{d}_j = \sum_{\mathsf{rvh}(x_j)=s} \mathrm{x}_j,$$

where we abbreviate $\mathsf{rvh}(t_0, t_f)$ by $\mathsf{rvh}(t_f)$ to ensure that the number of times s is used in the await loop is equal to the number of times s was generated as the return-value handler statement of some task. When $d_j \in \delta$ is a **post** transition from $T[\mathbf{post}\, r \leftarrow p\ e\ d]$ to $T[\mathbf{skip}]$, for some task context T, which creates a newly pending task $t = \langle e(T), s_p, d\rangle$, we write $d_j : t$. For each task $t \in \mathsf{Tasks}$, the program $\Phi_i(t_0, t_f)$ contains the constraint

$$\sum_{d_j:t} \mathrm{d}_j = \sum_{x_j=\langle t,\_\rangle} \mathrm{x}_j$$

to ensure that the number of times t is posted is equal to the number of times t's summaries are used.

Supposing $d_{j_1} d_{j_2} \ldots$ is a connected sequence of transitions through $\mathcal{A}_i$, a corresponding solution to the given set of constraints would set the variables $\{\mathrm{d}_j : j = j_1, j_2, \ldots\}$ to positive (nonzero) values corresponding to the number of times each transition is taken in $\mathcal{A}_i$. However, if there are loops in $\mathcal{A}_i$ which are not connected to any of the selected transitions, the given constraints do not prohibit solutions which take each transition of these loops an arbitrary number of times. This is a standard issue with encoding automaton traces which can be addressed by adding a polynomial number of constraints to the program $\Phi_i(t_0, t_f)$ [Seidl et al. 2004; Verma et al. 2005]. We take these additional constraints for granted. □

PROPOSITION 12.3. *For any two consecutive synchronization points $T_1[\mathbf{await}\, r]$ and $T_2[\mathbf{await}\, r]$ of $\mathcal{A}_P$, $\mathbf{0} \in \mathsf{sms}(T_1[\mathbf{skip}], T_2[\mathbf{skip}], \mathcal{A}_i)$ if and only if $\Phi_i(T_1[\mathbf{skip}], T_2[\mathbf{skip}])$ has a positive integer solution.*

Thus since the size of each ILP $\Phi_i(\cdot, \cdot)$ is polynomial in the size of $\mathcal{A}_i$, and thus in P, each synchronization point reachability query is decided in NP. As the set of possible

added transitions is bounded by $\left|\mathsf{Tasks}^2\right|$, our procedure thus terminates in polynomial time.

LEMMA 12.4. *The circuit satisfiability problem [Papadimitriou 1993] is polynomial-time reducible to the state-reachability problem for local-scope multi-wait single-region finite-value programs.*

PROOF. Let C be a Boolean circuit with wires W, gates G, inputs I, and an output wire $w_0 \in W$. Without loss of generality, assume that each gate $g \in G$ is connected to exactly two input wires and two output wires, and that each input $h \in I$ is connected to exactly two wires. The circuit satisfiability problem asks if there exists a valuation to the inputs I which makes the value of write w_0 true.

We construct a multi-wait single-region finite-value program P_C as follows. Let `Wire` be the type defined as

```
type Wire = { id: W, active: 𝔹, val: 𝔹 }
```

and define a procedure for writing a value to a wire,

```
proc set (var id: W, val: 𝔹)
    var w: Wire
    w.id := id;
    w.val := val;
    w.active := true;
    return (w,⋆),
```

which takes a value to be written and returns a wire, along with a nondeterministically chosen wire index; the return-value handler $\mathtt{d}_w(w,i)$ ensures[6] that the returned index i is either 1 or 2, assigns w to the `fst` wire variable, declared in the **init** procedure (which follows), when $i = 1$, and assigns w to the `snd` wire variable when $i = 2$.

For each gate $g \in G$ connected to input wires w_1, w_2, output wires w_3, w_4, and computing a function $f : \mathbb{B} \to \mathbb{B}$, we declare a procedure

```
proc p_g (var val: 𝔹)
    var fst0, snd0, fst, snd: Wire
    fst := fst0;
    snd := snd0;
    assume fst.active and fst.id = w_1;
    assume snd.active and snd.id = w_2;
    assume val = f(fst.val,snd.val);
    fst.active := false;
    snd.active := false;
    return (fst0,snd0,fst,snd),
```

which reads guessed values for the wires `fst` and `snd`, previously made active by the return-value handler $\mathtt{d}_w$ of two preceding `set` tasks; the return-value handler $\mathtt{d}_{\mathtt{rw}}(fst_0, snd_0, fst, snd)$ ensures that current values of the `fst` and `snd` wire variables declared in the **init** procedure have the values fst_0 and snd_0, and updates the `fst` and `snd` variables with the values fst and snd.

[6]We can block executions by allowing return handlers to be partial functions.

Finally, the initial procedure posts two instances of set per input $h \in I$, and two instances of set per gate $g \in G$, along with one instance of p_g, then waits until every task is consumed in some sequence

```
proc init ()
    var fst, snd: Wire
    var val: 𝔹
    var done: 𝔹

    fst.active := false;
    snd.active := false;
    done := false;

    // input h_1
    val := ⋆;
    post r ← set(w_{h_1,1},val) d_w;
    post r ← set(w_{h_1,2},val) d_w;

    // input h_2
    val := ⋆;
    post r ← set(w_{h_2,1},val) d_w;
    post r ← set(w_{h_2,2},val) d_w;

    ...;

    // gate g_1
    val := ⋆;
    post r ← p_{g_1}(val) d_rw;
    post r ← set(w_{g_1,3},val) d_w;
    post r ← set(w_{g_1,4},val) d_w;

    ...;

    await r;
    done := true,
```

where $w_{h_i j}$ (respectively, $w_{g_i j}$) denotes the jth wire of input h_i (respectively, gate g_i).

The program P_C simulates C by evaluating each gate $g \in G$ one-by-one at the **await** statement, based on an ordering such that g's input wires are active exactly when the task of procedure p_g is consumed. This is possible since the *setting* of each input wire $w \in W$ of g is also a pending task (of procedure set), which in turn can be scheduled immediately before p_g. Such an execution is guaranteed to be explored since every possible ordering of pending task consumption is considered at the **await** statement.

We then ask if there is a reachable state in which

$$\texttt{fst.id} = w_0 \textbf{ and } \texttt{fst.val} = \textbf{true} \textbf{ and } \texttt{done} = \textbf{true}$$

and if so, it must be the case that C is satisfiable. Inversely, if C is satisfiable then there must exist a corresponding execution of P_C since every possible circuit evaluation order is considered. □

12.2. Single-Region Multi-Wait Analysis

Here we demonstrate that the general problem of state reachability for finite-value single-region multi-wait programs is computationally equivalent to vector reachability

in vector addition systems, which is known to be EXPSPACE-hard [Lipton 1976], and decidable [Kosaraju 1982; Mayr 1981], though at present only nonprimitive recursive algorithms are known.

THEOREM 12.5. *The state-reachability problem for multi-wait single-region finite-value programs is decidable, and EXPSPACE-hard.*

We demonstrate this equivalence by a pair of polynomial-time reductions between our state-reachability problem, and configuration reachability in vector addition systems.

LEMMA 12.6. *The state-reachability problem for multi-wait finite-value programs is polynomial-time reducible to the reachability problem for vector addition systems.*

PROOF. Without the local-scoping restriction, each execution of each procedure $p \in \mathsf{Procs}$ between entry point $t_0 \in \mathsf{Tasks}$ and exit point $t_f \in \mathsf{Tasks}$ is summarized by the tasks posted between the last encountered **await** statement of p, at a so-called "synchronization point" $t_s \in \mathsf{Tasks}$—$t_s = t_0$ if no **await** statements are encountered before returning—and a **return** statement, at p's exit point t_f. Since p may make recursive procedure calls between any such reachable t_s and t_f, and each called procedure may again return pending tasks, the possible sets of pending tasks upon p's return at t_f is described by a context-free language $L(t_0, t_f)$. More precisely, since the task-posting actions between the last synchronization points and return points of p and each recursively called procedure commute, the set of pending tasks returned from p is a semilinear set, described by the Parikh-image of $L(t_0, t_f)$ [Parikh 1966]; this set can be characterized as the set of reachable vectors in a polynomially-sized vector addition system $W(t_0, t_f)$ without recursion and zero-test edges [Ganty and Majumdar 2012].

In what follows we will use such systems $W(t_0, t_f)$ as "widgets" [Ganty and Majumdar 2012], that is, components of an encompassing vector addition system, to compute a monotonically increasing sequence $E_0 \subseteq E_1 \subseteq \ldots$ of synchronization points summaries, similarly to the proof of Lemma 12.2. Initially, $E_0 = \emptyset$, and at each step $i > 0$ we add to E_i an edge $\langle T_1[\mathbf{skip}], T_2[\mathbf{skip}]\rangle$ summarizing a computation between consecutive synchronization points $T_1[\mathbf{await}\ r]$ and $T_2[\mathbf{await}\ r]$; an edge $\langle T_1[\mathbf{skip}], T_2[\mathbf{skip}]\rangle \in E_i$ implies that $\mathbf{0} \in \mathsf{sms}(T_1[\mathbf{skip}], T_2[\mathbf{await}\ r], \mathcal{A}_P)$, and the set of possible added edges is bounded, liberally by $\left|\mathsf{Tasks}^2\right|$.

To compute a new summary transition at step $i > 0$, we leverage Ganty and Majumdar's [2012] widgets to construct a set of VASSes $\{W_i(t_0, t_f) : t_0, t_f \in \mathsf{Tasks}\}$; each $W_i(t_0, t_f)$ computes the set of vectors returned by an execution of a given procedure from entry t_0 to exit t_f, using the summary transitions E_{i-1} between consecutive synchronization points. Using these widgets W_i, for a synchronization point summary query between t_1 and $t_2 = T[\mathbf{await}\ r]$, we construct a vector addition system $\mathcal{A}_i(t_1, t_2)$ which includes the intra-procedural and **post** transitions of $\mathcal{A}_P$, along with

$$T[\mathbf{await}\ r] \xrightarrow{\vec{n}_j \mathbf{0}} \langle q_0, T[\mathbf{skip}]\rangle \quad \text{and} \quad \langle q_f, T[\mathbf{skip}]\rangle \xrightarrow{\mathbf{00}} T[s; \mathbf{await}\ r]$$

for each $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and q_0 and q_f are the (assumed unique) initial and final states, respectively, of $W_i(t_0, t_f)$. Each pair of additional transitions, for $t_0, t_f \in \mathsf{Tasks}$, encode the consumption of a single task executing from t_0 to t_f by transitioning into and returning from $W_i(t_0, t_f)$. We build each transition of each $W_i(t_0, t_f)$ into $\mathcal{A}_i(t_1, t_2)$ by adding the transition

$$\langle q, t_2\rangle \xrightarrow{\vec{n}_1 \vec{n}_2} \langle q', t_2\rangle$$

Algorithm 2: A decision procedure for state reachability in multiwait programs. For simplicity we compute reachability to target valuations of the initial procedure frame.

Data: An initial condition ι and target valuation ℓ of a multi-wait program P
Result: Whether ℓ is reachable from ι in P

let $\mathcal{A}_P$ be the RVASS obtained from P;
let t_ι, t_ℓ be corresponding initial and target states of $\mathcal{A}_P$;
initialize E_0 to $\emptyset$;
initialize $\mathcal{A}_0$ to $\textsc{Compile}(P, E_0)$;
initialize i to 0;
while t_ℓ *is not reachable from* t_ι *in* $\mathcal{A}_i$ **do**
  compute $E_{i+1} := \{\langle T_1[\mathbf{skip}], T_2[\mathbf{skip}]\rangle : \mathbf{0} \in \mathsf{sms}(T_1[\mathbf{skip}], T_2[\mathbf{await}\ r], \mathcal{A}_i)\}$;
  if $E_{i+1} = E_i$ **then**
    return ℓ *is not reachable from* ι *in* P;
  end
  initialize $\mathcal{A}_{i+1}$ to $\mathcal{A}_i$;
  foreach $\langle t_1, t_2\rangle \in E_{i+1}$ **do**
    add the edge $\langle t_1, t_2\rangle$ to $\mathcal{A}_{i+1}$;
  end
  increment i;
end
return ℓ *is reachable from* ι *in* P;

whenever $q \xrightarrow{\vec{n}_1\vec{n}_2} q'$ is a transition of $W_i(t_0, t_f)$. Given the program P and a set E of synchronization point summaries, we write $\textsc{Compile}(P, E)$ to denote this system $\mathcal{A}$ obtained from $\mathcal{A}_P$ by replacing **await** transitions by transitions into and out from the widgets constructed with the summary edges E, as described before.

We ensure that at each step $i > 0$ the set E_i adds some edge which is not contained in E_{i-1}, so long as such a satisfiable query exists. Thus we reduce state reachability in $\mathcal{A}_P$ to several frame reachability queries in nonrecursive vector addition systems $\mathcal{A}_i$. Algorithm 2 lists our procedure precisely. □

PROPOSITION 12.7. *For any two consecutive synchronization points t_1 and t_2 of $\mathcal{A}_P$, $\mathbf{0} \in \mathsf{sms}(t_1, t_2, \mathcal{A}_P)$ if and only if there exists $k \in \mathbb{N}$ for $k < \left|\mathsf{Tasks}^2\right|$ such that $\mathbf{0} \in \mathsf{sms}(t_1, t_2, \mathcal{A}_k(t_1, t_2))$.*

The size of each VASS $\mathcal{A}_i(t_1, t_2)$ is polynomial in the size of P, and each synchronization point summary query $\langle t_1, t_2\rangle$ is decidable by reachability to $\mathbf{0}$ in $\mathcal{A}_i(t_1, t_2)$. As the set of possible queries is bounded (by $\left|\mathsf{Tasks}^2\right|$), our procedure is guaranteed to terminate.

Remark 12.8. Although we have considered only state reachability to a target state in the initial procedure frame, the same reasoning of Remark 10.3 applies here to generalize our algorithm to state reachability in an arbitrary procedure frame.

LEMMA 12.9. *The reachability problem for vector addition systems is polynomial-time reducible to the state-reachability problem for multi-wait finite-value programs.*

Note that the construction in the following proof requires only a nonrecursive (see Section 4) single-region program, with only a single **await** statement.

PROOF. Let $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ be a k-dimension vector addition system with $\hookrightarrow = \{d_1, \ldots, d_n\}$, and let $q_0, q_f \in Q$. Instead of checking reachability of a vector $\vec{n}_f$ from $\mathbf{0}$ in $\mathcal{A}$, we will instead solve an equally hard problem of checking whether $\mathbf{0}$ is reachable from an initial vector $\vec{n}_0$. To do this we construct a multi-wait program $P_{\mathcal{A}}$ and a local valuation ℓ which is reachable in $P_{\mathcal{A}}$ if and only if the configuration $q\mathbf{0}$ is reachable from $q\vec{n}_0$ in $\mathcal{A}$.

We will construct $P_{\mathcal{A}}$ such that the number of pending tasks in a configuration is equal to the sum of vector components in a corresponding configuration of $\mathcal{A}$. We then simulate each step of $\mathcal{A}$, which subtracts $\vec{n}_1 \in \mathbb{N}^k$ and adds $\vec{n}_2 \in \mathbb{N}^k$, by consuming $\sum_i \vec{n}_1(i)$ tasks and posting $\sum_i \vec{n}_2(i)$ tasks, while ensuring each task consumed (respectively, posted) corresponds to a subtraction (respectively, addition) to the correct vector component.

For each transition $d_i = \langle q, \vec{n}_1, \vec{n}_2, q \rangle$ we define the sequence $\sigma_i \in [1,k]^*$ of counter decrements as

$$\sigma_i \stackrel{\text{def}}{=} \underbrace{11\ldots11}_{\vec{n}_1(1) \text{ times}} \; \underbrace{22\ldots22}_{\vec{n}_1(2) \text{ times}} \; \ldots \; \underbrace{kk\ldots kk}_{\vec{n}_1(k) \text{ times}}.$$

We assume, without loss of generality, that each transition has a nonzero decrement vector, that is, $\vec{n}_1 \neq \mathbf{0}$ and thus $|\sigma_i| > 0$. We will use return-value handlers to ensure that a $|\sigma_i|$-length sequence of consecutively consumed tasks corresponds the decrement of transition d_i. For each $j \in \{1, \ldots, |\sigma_i|\}$, let $\mathtt{d}_{i,j}(\cdot)$ be the return-value handler defined by

```
assume cur_tx = i;
assume cur_pos = j;
cur_pos := cur_pos + 1;
```

which checks that consuming a given task corresponds to a decrement (by one) of the $\sigma_i(j)^{\text{th}}$ component of the decrement vector of $d_i \in \delta$. For each increment vector $\vec{n}$ (i.e., $\langle q, \vec{n}_1, \vec{n}, q \rangle \in \delta$ for some $\vec{n}_1 \in \mathbb{N}^k$), or initial vector $\vec{n} = \vec{n}_0$, we declare the procedure

```
proc inc_n⃗ ()
    for var idx := 1 to k do
        for var cnt := 1 to n⃗(idx) do
            let tx = ⋆
            and pos = ⋆ in
            assume σ_tx(pos) = idx;
            post r ← dummy() d_tx,pos,
along with the dummy procedure

    proc dummy ()
        return,
```

posting $\vec{n}(m)$ tasks for each $m \in \{1, \ldots, k\}$, to be consumed later by arbitrary positions j of the decrement sequences σ_i (since pos is assigned $\star$) of arbitrary transitions d_i (since tx is assigned $\star$) such that $\sigma_i(j) = m$—this ensures that the subsequent consumption of a task with handler $\mathtt{d}_{i,j}$ corresponds to decrementing the m^{th} component of $\vec{n}$.

The return-value handlers $\mathtt{d}_{i,j}$ corresponding to a given transition i with decrement vector $\vec{n}_1$ and increment vector $\vec{n}_2$ are forced to occur between the return-value handlers $\mathtt{d}_{i,\mathrm{b}}$ and $\mathtt{d}_{i,\mathrm{e}}$, defined by

```
assume cur_tx = ⊥;                 assume cur_tx = i;
cur_tx := i;                       assume cur_pos = |n⃗_1|+1;
cur_pos := 1          and          cur_tx := ⊥,
```

attached to the $\texttt{begin}_i$ and $\texttt{end}_i$ procedures

```
proc begin_i ()          proc end_i ()
    return                   call inc_{n⃗_2}.
```

Finally, the initial procedure `main` simply adds tasks corresponding to the initial vector $\vec{n}_0$ to an initially empty region container, and posts an arbitrary number of $\texttt{begin}_i$ and $\texttt{end}_i$ pairs, then waits until every task has been consumed.

```
proc main ()
    reg r
    var cur_tx := ⊥;
    var cur_pos;
    var empty := false;
    call inc_{n⃗_0} ();
    while ⋆ do
        let tx = ⋆ in
        post r ← begin_tx() d_{i,b};
        post r ← end_tx() d_{i,e};
    await r;

    // check: is this point reachable?
    empty := true;
    return
```

Checking that $P_{\mathcal{A}}$ faithfully simulates $\mathcal{A}$ is easily done by noticing the correspondence between configurations $q\vec{n}$ of $\mathcal{A}$ and configurations of $P_{\mathcal{A}}$ with $\sum_i \vec{n}$ pending tasks—excluding the $\texttt{begin}_i$ and $\texttt{end}_i$ tasks. Since `empty = true` is only reachable when there are no pending tasks, reachability to `empty = true` implies $\mathbf{0}$ is reachable in $\mathcal{A}$. Furthermore, if $\mathbf{0}$ is reachable in $\mathcal{A}$, a run of $P_{\mathcal{A}}$ will eventually proceed past the **await** statement without pending tasks, setting `empty = true`. □

PROPOSITION 12.10. *The configuration $q\mathbf{0}$ is reachable in $\mathcal{A}$ from $q\vec{n}_0$ if and only if* `empty = true` *is reachable in $P_{\mathcal{A}}$.*

Since the size of $P_{\mathcal{A}}$ is polynomial in $\mathcal{A}$, we have a polynomial-time reduction for deciding configuration reachability in $\mathcal{A}$.

Since finding practical algorithms to compute vector reachability is a difficult open problem, we remark that it is possible to obtain algorithms to approximate our state-reachability problem. Consider, for instance, the overapproximate semantics given by transforming each **await** r statement into **while** $\star$ **do ewait** r. Though many more behaviors are present in the resulting program, since not every task is necessarily consumed during the **while** loop, having practical algorithmic solutions is perhaps more likely (see Section 9.4).

13. RELATED WORK

Formal modeling and verification of multithreaded programs has been heavily studied, including but not limited to identifying decidable subclasses [Kahlon 2009], and effective overapproximate [Elmas et al. 2009; Flanagan and Qadeer 2003; Gupta et al. 2011; Henzinger et al. 2003; Owicki and Gries 1976] and underapproximate analyses [Atig et al. 2011; Emmi et al. 2011; Esparza and Ganty 2011; Kahlon 2009; Lal and Reps 2009; LaTorre et al. 2009, 2010; Musuvathi and Qadeer 2007; Qadeer and Wu 2004; Qadeer and Rehof 2005].

State Reachability in Recursively Parallel Programs			
	result	complexity	language/feature
Task-Passing			
general	Thm. 4.3	undecidable	futures, revisions
Single-Wait			
non-aliasing	Thm. 9.2	PTIME	futures†, revisions†
local scope	Thm. 9.5	EXPSPACE	—
global scope	Thm. 9.9	EXPSPACE	asynchronous programs
general	Thm. 9.13	2EXPTIME	—
† For programs without bidirectional task-passing.			
Multi-Wait (single region)			
local scope	Thm. 12.1	NP	Cilk, parallel for-each
general	Thm. 12.5	decidable	async (X10)

Fig. 18. Summary of results for computing state-reachability for finite-value recursively parallel programs.

To our knowledge little work has been done in formal modeling and verification of programs written in explicitly parallel languages which are free of thread interleaving. Though Lee and Palsberg [2010] have developed a core calculus for an X10-like language with **async/finish** statements, their calculus does not handle the variety of semantics we have covered, nor had they studied complexity, or made the connection with a fundamental formal model, such as VASS. Sen and Viswanathan's [2006] asynchronous programs, which falls out as a special case of our single-wait programs, is perhaps most similar to our work. Practical verification algorithms for safety [Jhala and Majumdar 2007] and liveness [Ganty et al. 2009], and in-depth complexity analysis [Ganty and Majumdar 2012] of asynchronous programs have been studied, as well as extensions to add task priorities [Atig et al. 2008] or ordering and synchronization [Geeraerts et al. 2012], and underapproximate analyses based on systematically limiting nondeterminism [Emmi et al. 2011, 2012].

Though here we focus on the problem of state-reachability as a general means of establishing safety properties in parallel programs, others have studied pairwise reachability [Kahlon 2009], also known as "may-happen-in-parallel analysis." After Taylor [1983] demonstrated the intractability of computing pairwise reachability, Agarwal et al. [2007] and Lee and Palsberg [2010] developed overapproximating constraint-based type systems to establish whether two procedures may happen in parallel in X10. Others have studied predominantly polynomial-time bitvector and constraint-propagation problems [Esparza and Podelski 2000; Knoop et al. 1996; Seidl and Steffen 2000], whose structure leads to more tractable analyses than the more general state-reachability problem.

Though decidability results of abstract parallel models have been reported [Bouajjani et al. 2005; Esparza and Podelski 2000] (Bouajjani and Esparza [2006] survey of this line of work), these works target abstract computation models, and do not identify precise complexities and optimal algorithms for real-world parallel programming languages, nor do they handle the case where procedures can return unbounded sets of unfinished computations to their callers.

14. CONCLUSION

We have proposed a general model of recursively parallel programs which captures the concurrency constructs in a variety of popular parallel programming languages. By isolating the fragments corresponding to various language features, we are able to associate corresponding formal models, measure the complexity of state reachability, and provide precise analysis algorithms. We hope our complexity measurements may be used to guide the design and choice of concurrent programming languages and program analyses. Figure 18 summarizes our results.

ACKNOWLEDGMENTS

We greatly appreciate formative discussions with Arnaud Sangnier and Peter Habermehl, and the feedback of Pierre Ganty, Giorgio Delzanno, Rupak Majumdar, Tom Ball, Sebastian Burckhardt, Sylvain Schmitz, Alain Finkel, and the anonymous POPL and TOPLAS reviewers.

REFERENCES

Abdulla, P. A., Cerans, K., Jonsson, B., and Tsay, Y.-K. 1996. General decidability theorems for infinite-state systems. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS'96)*. 313–321.

Agarwal, S., Barik, R., Sarkar, V., and Shyamasundar, R. K. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'07)*. ACM Press, New York, 183–193.

Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. 2006. The fortress language specification. Tech. rep., Sun Microsystems, Inc.

Apt, K. R. and Kozen, D. 1986. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett. 22,* 6, 307–309.

Atig, M. F., Bouajjani, A., and Touili, T. 2008. Analyzing asynchronous programs with preemption. In *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*. Leibniz International Proceedings in Informatics, vol. 2, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 37–48.

Atig, M. F., Bouajjani, A., and Qadeer, S. 2011. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logic. Methods Comput. Sci. 7,* 4.

Ball, T., Chaki, S., and Rajamani, S. K. 2001. Parameterized verification of multithreaded software libraries. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*. Lecture Notes in Computer Science, vol. 2031, Springer, 158–173.

Bouajjani, A. and Esparza, J. 2006. Rewriting models of Boolean programs. In *Proceedings of the 17th International Conference on Term Rewriting and Applications (RTA'06)*. Lecture Notes in Computer Science, vol. 4098, Springer, 136–150.

Bouajjani, A., Müller-Olm, M., and Touili, T. 2005. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*. Lecture Notes in Computer Science, vol. 3653, Springer, 473–487.

Burckhardt, S., Baldassin, A., and Leijen, D. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. ACM Press, New York, 691–707.

Chamberlain, B. L., Callahan, D., and Zima, H. P. 2007. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl. 21,* 3, 291–312.

Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM Press, New York, 519–538.

Chaudhuri, S. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, 159–169.

Clarke, E. M. and Emerson, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, Lecture Notes in Computer Science, vol. 131, Springer, 52–71.

Clarke, E. M., Grumberg, O., and Peled, D. 2001. *Model Checking*. MIT Press.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*. 106–130.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, New York, 238–252.

Delzanno, G., Raskin, J.-F., and Begin, L. V. 2002. Towards the automated verification of multithreaded Java programs. In *Proceedings of tghe 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*. Lecture Notes in Computer Science, vol. 2280, Springer, 173–187.

Demri, S., Jurdzinski, M., Lachish, O., and Lazic, R. 2009. The covering and boundedness problems for branching vector addition systems. In *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'09)*. Leibniz International Proceedings in Informatics Series, vol. 4, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 181–192.

Elmas, T., Qadeer, S., and Tasiran, S. 2009. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, New York, 2–15.

Emmi, M., Qadeer, S., and Rakamaric, Z. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 411–422.

Emmi, M., Lal, A., and Qadeer, S. 2012. Asynchronous programs with prioritized task-buffers. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM Press, New York.

Esparza, J. and Nielsen, M. 1994. Decidability issues for Petri nets - A survey. *Bull. Euro. Assoc. Theor. Comput. Sci. 52*, 244–262.

Esparza, J. and Podelski, A. 2000. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proceedings of thge 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*. ACM Press, New York, 1–11.

Esparza, J. and Ganty, P. 2011. Complexity of pattern-based verification for multithreaded programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 499–510.

Esparza, J., Kucera, A., and Schwoon, S. 2003. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput. 186,* 2, 355–376.

Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (SC'06)*. ACM Press, New York, 83.

Finkel, A. and Schnoebelen, P. 2001. Well-structured transition systems everywhere! *Theor. Comput. Sci. 256,* 1–2, 63–92.

Flanagan, C. and Felleisen, M. 1999. The semantics of future and an application. *J. Funct. Program. 9,* 1, 1–31.

Flanagan, C. and Qadeer, S. 2003. Thread-modular model checking. In *Proceedings of the 10th International Workshop on Model Checking Software (SPIN'03)*. Lecture Notes in Computer Science, vol. 2648, Springer, 213–224.

Ganty, P. and Majumdar, R. 2012. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst. 34,* 1, 6.

Ganty, P., Majumdar, R., and Rybalchenko, A. 2009. Verifying liveness for asynchronous programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, New York, 102–113.

Geeraerts, G., Heußner, A., and Raskin, J.-F. 2012. Queue-dispatch asynchronous systems. CoRR abs/1201.4871. http://www.swt-bamberg.de/aheussner/files/heussner-a-2013–a.pdf.

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

German, S. M. and Sistla, A. P. 1992. Reasoning about systems with many processes. *J. ACM 39,* 3, 675–735.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*. Lecture Notes in Computer Science Series, vol. 1254, Springer, 72–83.

Gray, J. 1985. Why do computers stop and what can be about it? http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

Gupta, A., Popeea, C., and Rybalchenko, A. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 331–344.

Halstead Jr., R. H. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4, 501–538.

Henzinger, T. A., Jhala, R., Majumdar, R., and Qadeer, S. 2003. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*. Lecture Notes in Computer Science, vol. 2725, Springer, 262–274.

Jhala, R. and Majumdar, R. 2007. Interprocedural analysis of asynchronous programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. ACM Press, New York, 339–350.

Kahlon, V. 2009. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS'09)*. 27–36.

Karp, R. M. and Miller, R. E. 1969. Parallel program schemata. *J. Comput. Syst. Sci. 3,* 2, 147–195.

Knoop, J., Steffen, B., and Vollmer, J. 1996. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst. 18,* 3, 268–299.

Kosaraju, S. R. 1982. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82)*. ACM Press, New York, 267–281.

Kozen, D. 1977. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*. IEEE Computer Society, Lois Alamitos, CA, 254–266.

La Torre, S., Madhusudan, P., and Parlato, G. 2009. Reducing context-bounded concurrent reachability to sequential reachability. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*. Lecture Notes in Computer Science, vol. 5643, Springer, 477–492.

La Torre, S., Madhusudan, P., and Parlato, G. 2010. Model-checking parameterized concurrent programs using linear interfaces. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. Lecture Notes in Computer Science, vol. 6174, Springer, 629–644.

Lal, A. and Reps, T. W. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods Syst. Des. 35,* 1, 73–97.

Larus, J. R. and Rajwar, R. 2006. *Transactional Memory*. Morgan & Claypool. http://www.morganclaypool.com/doi/abs/10.2200/S00070ED1V01Y200611CAC002.

Lee, E. A. 2006. The problem with threads. *IEEE Comput. 39,* 5, 33–42.

Lee, J. K. and Palsberg, J. 2010. Featherweight X10: A core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'10)*. ACM Press, New York, 25–36.

Leijen, D., Schulte, W., and Burckhardt, S. 2009. The design of a task parallel library. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*. ACM Press, New York, 227–242.

Leroux, J. 2011. Vector addition system reachability problem: A short self-contained proof. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, New York, 307–316.

Lipton, R. J. 1976. The reachability problem requires exponential space. Tech. rep. 62, Yale University.

Mayr, E. W. 1981. An algorithm for the general Petri net reachability problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*. ACM Press, New York, 238–246.

Musuvathi, M. and Qadeer, S. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM Press, New York, 446–455.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of program analysis*. Springer.

Owicki, S. S. and Gries, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM 19,* 5, 279–285.

Papadimitriou, C. H. 1993. *Computational Complexity*. Addison Wesley.

Parikh, R. 1966. On context-free languages. *J. ACM 13,* 4, 570–581.

Pratikakis, P., Vandierendonck, H., Lyberis, S., and Nikolopoulos, D. S. 2011. A programming model for deterministic task parallelism. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'11)*. ACM Press, New York, 7–12.

Qadeer, S. and Wu, D. 2004. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM Press, New York, 14–24.

Qadeer, S. and Rehof, J. 2005. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. Lecture Notes in Computer Science, vol. 3440, Springer, 93–107.

Queille, J.-P. and Sifakis, J. 1982. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium of International Symposium on Programming*. Lecture Notes in Computer Science, vol. 137, Springer, 337–351.

Rackoff, C. 1978. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci. 6*, 223–231.

Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22,* 2, 416–430.

Randall, K. H. 1998. Cilk: Efficient multithreaded computing. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Reps, T. W., Horwitz, S., and Sagiv, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM Press, New York, 49–61.

Savitch, W. J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci. 4,* 2, 177–192.

Segulja, C. and Abdelrahman, T. S. 2011. Synchronization-free and deterministic coarse-grain parallelism: Architectural support and programming model. In *Proceedings of the 1st International Workshop on Future Architectural Support for Parallel Programming*.

Seidl, H. and Steffen, B. 2000. Constraint-based inter-procedural analysis of parallel programs. In *Proceedings of the 9th European Symposium on Programming (ESOP'00)*. Lecture Notes in Computer Science, vol. 1782, Springer, 351–365.

Seidl, H., Schwentick, T., Muscholl, A., and Habermehl, P. 2004. Counting in trees for free. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*. Lecture Notes in Computer Science, vol. 3142, Springer, 1136–1149.

Sen, K. and Viswanathan, M. 2006. Model checking multithreaded programs with asynchronous atomic methods. In *Proceedings 18th International Conference on Computer Aided Verification (CAV'06)*. Lecture Notes in Computer Science, vol. 4144, Springer, 300–314.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data-flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones Eds., Prentice-Hall, Chapter 7, 189–234.

Sipser, M. 1997. *Introduction to the theory of computation*. PWS Publishing Company.

Taylor, R. N. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf. 19*, 57–84.

Verma, K. N. and Goubault-Larrecq, J. 2005. Karp-Miller trees for a branching extension of VASS. *Discr. Math. Theor. Comput. Sci. 7,* 1, 217–230.

Verma, K. N., Seidl, H., and Schwentick, T. 2005. On the complexity of equational horn clauses. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*. Lecture Notes in Computer Science, vol. 3632, Springer, 337–352.

Received August 2012; revised February 2013; accepted May 2013