Open access • Journal Article • DOI:10.1145/2518188

Analysis of Recursively Parallel Programs — Source link

Ahmed Bouajjani, Michael Emmi

Institutions: Paris Diderot University

Published on: 01 Nov 2013 - ACM Transactions on Programming Languages and Systems (ACM)

Topics: Cilk, EXPSPACE, Decidability, Concurrency and Undecidable problem

Analysis of Recursively Parallel Programs

Ahmed Bouajjani, Michael Emmi

To cite this version:

Ahmed Bouajjani, Michael Emmi. Analysis of Recursively Parallel Programs. 2011. hal-00639351v2

Analysis of Recursively Parallel Programs [*]

Ahmed Bouajjani
LIAFA, Université Paris Diderot, France
abou@liafa.jussieu.fr

Michael Emmi [†]
LIAFA, Université Paris Diderot, France
mje@liafa.jussieu.fr

Abstract

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 multi-threaded programs.

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 shared memory. Such nondeterminism leads to rarely-occurring "Heisenbugs" which are notoriously difficult to reproduce and repair. To prevent such bugs programmers are faced with the difficult task of preventing undesirable interleavings, e.g., by employing lock-based synchronization, without preventing benign interleavings—otherwise the desired reactivity or parallelism is forfeited.

The complexity of multi-threaded program analysis seems to comply with the perceived difficulty of multi-threaded programming. The state-reachability problem for multi-threaded programs is PSPACE-complete [21] with a finite number of finite-state threads, and undecidable [30] with recursive threads. Current analysis approaches either explore an underapproximate concurrent semantics by considering relatively few interleavings [9, 22] or explore a coarse overapproximate semantics via abstraction [13, 18].

Explicitly-parallel programming languages have been advocated to avoid the intricate interleavings implicit in program syntax [24], and several such industrial-strength languages have been developed [2, 5, 6, 17, 25, 31, 33]. Such systems introduce various mechanisms for creating (e.g., **fork**, **spawn**, **post**) and 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 [6], data-replication [5], functional programming [17], message passing [28], or version-based memory access models [33], perhaps falling back on transactional mechanisms [23] when complete isolation is impractical. Although few 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 allow computations creating an unbounded number of sub-computations, returning to their superiors an unbounded number of handles to unfinished 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 program reasoning, how those models compare with each other, and how to design appropriate analysis algorithms. We attempt to answer these questions as follows:

- We introduce a general interleaving-free parallel programming model on which to express the features found in popular parallel programming languages (Section 2).
- We discover a surprisingly-complex feature of some existing languages: even simple classes of programs with the ability to pass unfinished computations both to and from subordinate computations have undecidable state-reachability problems (Section 2.4).
- We show that the concurrency features present in many real-world programming languages such as Cilk, X10, and Multilisp are captured precisely (modulo the possibility of interleaving) by various fragments of our model (Sections 4 and 6).
- For fragments corresponding to real-world language features, we measure the complexity of computing state-reachability for finite-data programs, and provide, in most cases, asymptotically optimal state-reachability algorithms (Sections 5 and 7).

Our focus on finite-data programs without interleaving is a means to measuring complexity for the sake of comparison, required since state-reachability for infinite-data or multi-threaded programs is generally undecidable. Applying our algorithms in practice may rely on data abstraction [16], and separately ensuring isolation [23], or approximating possible interleavings [9, 13, 18, 22]; still, our handling of computation-order non-determinism is precise.

The major distinguishing language features are whether a single or an arbitrary number of subordinate computations are waited for at once, and whether the scope of subordinate computations is confined. Generally speaking, reasoning for the "single-wait" case of Section 4 is less difficult than for the "multi-wait" case of Section 6, and we demonstrate a range of complexities[1] from PTIME, NP, EXPSPACE, and 2EXPTIME for various scoping restrictions in Sections 5 and 7. Despite these worst-case complexities, a promising line of work has

---

[*] Partially supported by the project ANR-09-SEGI-016 Veridyc.

[†] Supported by a post-doctoral fellowship from the Fondation Sciences Mathématiques de Paris.

[0] Proofs to technical results are contained in the appendices.

[1] In order to isolate concurrent complexity from the exponential factor in the number of program variables, we consider a fixed number of variables in each procedure frame; this allows us a PTIME point-of-reference for state-reachability in recursive sequential programs [32].

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

Figure 1. 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.

already demonstrated effective algorithms for practically-occurring EXPSPACE-complete state-reachability problem instances based on simultaneously computing iterative under- and over-approximations, and rapidly converging to a fixed point [15, 19].

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. 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 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—i.e., 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 unconsumed tasks.

This model permits vastly 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.

2.1 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 $\mathsf{Rets} \subseteq (\mathsf{Vals} \rightarrow \mathsf{Stmts})$ a set of return-value handlers. The grammar of Figure 1 describes our language of *recursively parallel programs*. We intentionally leave the syntax of expressions e unspecified, though we do insist Vals contains $\mathbf{true}$ and $\mathbf{false}$, 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 $n \in \mathbb{N}$ (resp., 1) region identifiers as *n-region* (resp., *single-region*) programs. A *sequential program* is a program without $\mathbf{post}$, $\mathbf{ewait}$, and $\mathbf{await}$ statements.

Each program P declares a sequence of procedures named $p_0 \dots p_i \in \mathsf{Procs}^*$, 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. Intuitively, a $\mathbf{post}\ r \leftarrow p\ e\ \vec{r}\ d$ 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 a subset of the parent's task handles in regions $\vec{r}$, and a return-value handler d. The $\mathbf{ewait}\ r$ 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 $\mathbf{await}\ r$ 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 $\mathbf{call}$, $\mathbf{return}$, $\mathbf{post}$, $\mathbf{ewait}$ and $\mathbf{await}$ 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 $\mathbf{assume}\ e$ statement proceeds only when e evaluates to $\mathbf{true}$—we use this statement in subsequent sections to block undesired executions in our encodings of other parallel programming models.

Example 1. The Fibonacci function can be implemented as a single-region recursively parallel program as follows.

```
proc fib (var n: ℕ)
   var sum: ℕ
   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
```

Alternate implementations are possible, e.g., by replacing the $\mathbf{await}$ statement by two $\mathbf{ewait}$ 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 $\mathbf{await}$ statement before returning).

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 variable per procedure.

2.2 Parallel Semantics with Task-Passing

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, creating a new frame by posting a task, or removing a frame by consuming a completed task; unconsumed sub-task frames of a completed task are added as children to the completed task's parent.

A *task* $\langle \ell, s, d \rangle$ is a valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable $\mathtt{l}$, 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* $m : \mathsf{Regs} \rightarrow \mathbb{M}[\mathsf{Configs}]$: for $r \in \mathsf{Regs}$, the multiset $m(r)$ specifies 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: $(m_1 \cup m_2)(r) \overset{\text{def}}{=} m_1(r) \cup m_2(r)$ for all $r \in \mathsf{Regs}$. The projection $m\mid_{\vec{r}}$ of a region valuation m to a region sequence $\vec{r}$ is defined by $m\mid_{\vec{r}}(r') = m(r')$ when r' occurs in $\vec{r}$, and $m\mid_{\vec{r}}(r') = \emptyset$ 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

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

—as $\mathtt{l}$ is the only variable, the expression $e[\ell/\mathtt{l}]$ 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 effect 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 $\mathsf{rvh}(t) \overset{\text{def}}{=} \{d(v) : v \in e(\ell)\}$ as the set of possible return-value handler statements for t; $\mathsf{rvh}(t)$ is undefined when t is not completed.

Figure 2 and Figure 3 define the transition relation $\to^{\mathrm{rpp/p}}$ of recursively parallel programs as a set of operational steps on configurations. The intra-procedural transitions $\to^{\mathrm{seq}}$ of individual tasks in Figure 2 are standard. More interesting are the inter-procedural transitions of Figure 3, which implicitly include a transition $C[\langle t_1, m \rangle] \to_P^{\mathrm{rpp/p}} C[\langle t_2, m \rangle]$ whenever $t_1 \to_P^{\mathrm{seq}} t_2$. The POST-T rule creates a procedure frame to execute in parallel, and links it to the current frame by the given region, passing ownership of tasks in the specified region sequence to the newly-created frame. The $\exists$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 $\forall$WAIT-NEXT-T and $\forall$WAIT-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

$$\begin{array}{l} \textbf{post}\ \mathtt{r}_{call} \leftarrow p\ e\ \varepsilon\ \mathtt{d}_{call}\,; \\ \textbf{ewait}\ \mathtt{r}_{call}\,, \end{array}$$

where $\mathtt{d}_{call}(v) \overset{\text{def}}{=} \mathtt{l}\ \mathtt{:=}\ v$ is the return-value handler which simply writes the entire return value v into the local variable $\mathtt{l}$, and 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 $c_i \to_P^{\mathrm{rpp/p}} c_{i+1}$ for $0 \leq i < j$. 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.

$$\textsc{Post-T}\quad \frac{v \in e(T) \qquad m' = m \setminus m\mid_{\vec{r}} \cup \left(r \mapsto \left\langle \langle v, s_p, d \rangle, m\mid_{\vec{r}} \right\rangle\right)}{C[\langle T[\textbf{post}\ r \leftarrow p\ e\ \vec{r}\ d], m \rangle] \xrightarrow[P]{\mathrm{rpp/p}} C\left[\left\langle T[\textbf{skip}], m' \right\rangle\right]}$$

$$\exists\textsc{wait-T}\quad \frac{m_1 = (r \mapsto \langle t_2, m_2 \rangle) \cup m_1' \qquad s \in \mathsf{rvh}(t_2)}{C[\langle T_1[\textbf{ewait}\ r], m_1 \rangle] \xrightarrow[P]{\mathrm{rpp/p}} C\left[\left\langle T_1[s], m_1' \cup m_2 \right\rangle\right]}$$

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

$$\forall\textsc{wait-Done-T}\quad \frac{m(r) = \emptyset}{C[\langle T[\textbf{await}\ r], m \rangle] \xrightarrow[P]{\mathrm{rpp/p}} C[\langle T[\textbf{skip}], m \rangle]}$$

Figure 3. The tree-based transition relation for parallely-executing recursively parallel programs with task-passing.

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.*

2.3 Sequential Semantics with Task-Passing

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 and **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 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 2.2, i.e., a finite unordered tree of task-labeled vertices and region-labeled edges. (Here all non-root 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 2 and 4 define the sequential transition relation $\to^{\mathrm{rpp/s}}$ of recursively parallel programs as a set of operational steps on configurations. The inter-procedural transitions of Figure 4 implicitly include a transition $\langle t_1, m \rangle\, c \to_P^{\mathrm{rpp/s}} \langle t_2, m \rangle\, c$ whenever $t_1 \to_P^{\mathrm{seq}} t_2$. 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

SKIP

$$\frac{}{T[\mathbf{skip};\ s] \xrightarrow[P]{\text{seq}} T[s]}$$

ASSUME

$$\frac{\mathbf{true} \in e(T)}{T[\mathbf{assume}\ e] \xrightarrow[P]{\text{seq}} T[\mathbf{skip}]}$$

IF-THEN

$$\frac{\mathbf{true} \in e(T)}{T[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \xrightarrow[P]{\text{seq}} T[s_1]}$$

IF-ELSE

$$\frac{\mathbf{false} \in e(T)}{T[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \xrightarrow[P]{\text{seq}} T[s_2]}$$

ASSIGN

$$\frac{\ell' \in e(\ell)}{\langle \ell, S[\mathtt{l\ :=\ }e], d\rangle \xrightarrow[P]{\text{seq}} \langle \ell', S[\mathbf{skip}], d\rangle}$$

LOOP-DO

$$\frac{\mathbf{true} \in e(T)}{T[\mathbf{while}\ e\ \mathbf{do}\ s] \xrightarrow[P]{\text{seq}} T[s; \mathbf{while}\ e\ \mathbf{do}\ s]}$$

LOOP-END

$$\frac{\mathbf{false} \in e(T)}{T[\mathbf{while}\ e\ \mathbf{do}\ s] \xrightarrow[P]{\text{seq}} T[\mathbf{skip}]}$$

Figure 2. The intra-procedural transition relation for recursively parallel programs.

POST-S

$$\frac{v \in e(T) \qquad m' = m \setminus m\,|_{\vec{r}} \cup \left(r \mapsto \left\langle \langle v, s_p, d\rangle, m\,|_{\vec{r}} \right\rangle\right)}{\langle T[\mathbf{post}\ r \leftarrow p\ e\ \vec{r}\ d], m\rangle\, c \xrightarrow[P]{\text{rpp/s}} \langle T[\mathbf{skip}], m'\rangle\, c}$$

∃WAIT-S

$$\frac{m = (r \mapsto c_0) \cup m'}{\langle T[\mathbf{ewait}\ r], m\rangle\, c \xrightarrow[P]{\text{rpp/s}} c_0\, \langle T[\mathbf{skip}], m'\rangle\, c}$$

∀WAIT-NEXT-S

$$\frac{m = (r \mapsto c_0) \cup m'}{\langle T[\mathbf{await}\ r], m\rangle\, c \xrightarrow[P]{\text{rpp/s}} c_0\, \langle T[\mathbf{skip};\ \mathbf{await}\ r], m'\rangle\, c}$$

∀WAIT-DONE-S

$$\frac{m(r) = \emptyset}{\langle T[\mathbf{await}\ r], m\rangle\, c \xrightarrow[P]{\text{rpp/s}} \langle T[\mathbf{skip}], m\rangle\, c}$$

RETURN-S

$$\frac{s \in \mathsf{rvh}(t_1)}{\langle t_1, m_1\rangle\, \langle T_2[\mathbf{skip}], m_2\rangle\, c \xrightarrow[P]{\text{rpp/s}} \langle T_2[s], m_1 \cup m_2\rangle\, c}$$

Figure 4. The stack-based transition relation for sequentially-executing recursively parallel programs with task-passing.

local valuation. The definitions of *sequential execution*, *initial*, and *reachable* are nearly identical to their parallel counterparts.

Lemma 1. *The parallel semantics and the sequential semantics are indistinguishable w.r.t. state reachability, i.e., 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.*

2.4 Undecidability of State-Reachability with Task-Passing

Recursively parallel programs allow pending tasks to be passed *bidirectionally*: both from completed tasks and to newly-created tasks. This capability makes the state-reachability problem undecidable—even for the very simple cases recursive programs with at least one region, and for non-recursive programs with at least two regions. 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 most-recently created task, after having passed the handle of the previously-most-recently created task to the most-recently created task. We can then show that such unbounded chains of pending tasks can be used to simulate an arbitrary unbounded and ordered storage device.

Definition 1 (Task passing)**.** A program which contains a statement $\mathbf{post}\ r \leftarrow p\ e\ \vec{r}\ d$, such that $|\vec{r}| > 0$ is called *task-passing*.

The *task-depth* of a program P is the maximum length of a sequence $p_1 \ldots p_i$ of procedures in P such that each p_j contains a statement $\mathbf{post}\ r \leftarrow p_{i+j}\ e\ \vec{r}\ d$, 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 *non-recrusive*.

Theorem 1. *The state-reachability problem for n-region finite-value task-passing parallel programs is undecidable for*

(a) non-recursive programs with $n > 1$, and
(b) recursive programs with $n > 0$.

The proof of Theorem 1 is given by two separate reductions from the emptiness problem for Turing machines to "single-wait" programs, i.e., those using **ewait** statements but not **await** statements. 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. When only one region is granted and recursion is not allowed, neither of these 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.

3. Programs without Task Passing

Due to the undecidability result of Theorem 1 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 $\mathbf{post}\ \ r\ \leftarrow\ p\ \ e\ \ d$. 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.

3.1 Sequential Semantics without Task-Passing

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. The transition relation $\to^{\text{rpp}}$ of Figures 2 and 5 implicitly include a transition $\langle t_1, m\rangle\, c \to_P^{\text{rpp}} \langle t_2, m\rangle\, c$ whenever $t_1 \to_P^{\text{seq}} t_2$. 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 2. *For all initial conditions ι non-task-passing programs P, the valuation ℓ is reachable in P from ι by a sequential execution*

Post

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

∃wait

$$\frac{m = (r \mapsto t_2) \cup m'}{\langle T_1[\textbf{ewait}\ r]\,, m \rangle\ c \xrightarrow[P]{\text{rpp}} \langle t_2, \emptyset \rangle\ \langle T_1[\textbf{skip}]\,, m' \rangle\ c}$$

∀wait-Next

$$\frac{m = (r \mapsto t_2) \cup m'}{\langle T_1[\textbf{await}\ r]\,, m \rangle\ c \xrightarrow[P]{\text{rpp}} \langle t_2, \emptyset \rangle\ \langle T_1[\textbf{skip};\ \textbf{await}\ r]\,, m' \rangle\ c}$$

∀wait-Done

$$\frac{m(r) = \emptyset}{\langle T[\textbf{await}\ r]\,, m \rangle\ c \xrightarrow[P]{\text{rpp}} \langle T[\textbf{skip}]\,, m \rangle\ c}$$

Return

$$\frac{s \in \mathsf{rvh}(t_1)}{\langle t_1, m_1 \rangle\ \langle T_2[\textbf{skip}]\,, m_2 \rangle\ c \xrightarrow[P]{\text{rpp}} \langle T_2[s]\,, m_1 \cup m_2 \rangle\ c}$$

Figure 5. The stack-based transition relation for sequentially-executing recursively parallel programs without task-passing.

with task-passing if and only if ℓ is reachable in P from ι by a sequential execution without task-passing.

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 the program model which correspond to the concurrency mechanisms found in real-world parallel programming languages.

3.2 Recursive Vector Addition Systems with Zero-Test Edges

Fix $k \in \mathbb{N}$. A *recursive vector addition system (RVASS)* $\mathcal{A} = \langle Q, \delta \rangle$ *of dimension* k is a finite set Q of states, along with a finite set $\delta = \delta_1 \uplus \delta_2 \uplus \delta_3$ of transitions partitioned into *additive* transitions $\delta_1 \subset Q \times \mathbb{N}^k \times \mathbb{N}^k \times Q$, *recursive* transitions $\delta_2 \subseteq Q \times Q \times Q \times Q$, and *zero-test* transitions $\delta_3 \subseteq Q \times Q$. We write

$$\begin{aligned}
q \overset{\vec{n}_1 \vec{n}_2}{\hookrightarrow} q' \qquad & \text{when } \langle q, \vec{n}_1, \vec{n}_2, q' \rangle \in \delta_1, \text{ and} \\
q \overset{q_1 q_2}{\hookrightarrow} q' \qquad & \text{when } \langle q, q_1, q_2, q' \rangle \in \delta_2. \\
q \hookrightarrow q' \qquad & \text{when } \langle q, q' \rangle \in \delta_3.
\end{aligned}$$

A *(non-recursive) vector addition system (with states) (VASS)* is a recursive vector addition system $\langle Q, \delta \rangle$ such that δ 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 non-empty sequence of frames representing a stack of non-recursive sub-computations. The transition relation $\to^{\text{rvas}}$ for recursive vector addition systems is defined in Figure 6. 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 (resp., Return) rule as a *call (resp., return) transition*. Finally, the Zero rule proceeds only when the top-most frame's vector equals $\mathbf{0}$.

An *execution of a RVASS $\mathcal{A}$ (from c_0 to c_j)* is a configuration sequence $c_0 c_1 \ldots c_j$ where $c_i \to^{\text{rvas}} c_{i+1}$ for $0 \leq i < j$. A configuration $\langle q, \vec{n} \rangle$ is called q_0*-initial* when $q = q_0$ and $\vec{n} = \mathbf{0}$, and a configuration c_f is called q_f*-final* when $c_f = \langle q_f, \vec{n} \rangle\ c$ for some configuration c and $\vec{n} \in \mathbb{N}^k$. We say a state q_f is *reachable in* $\mathcal{A}$ from q_0 when there exists an execution of $\mathcal{A}$ from some q_0-initial configuration c_0 to some q_f-final configuration c_f. The *state-reachability problem* for recursive vector addition systems is to determine whether a given state q is reachable from some q_0.

Recently Demri et al. [8] have proved that state-reachability in branching vector addition systems (BVAS)—a very similar formal model to which RVASS reduces—is in 2EXPTIME. This immediately gives us an upper-bound on computing state-reachability in RVASS without zero-test edges. Though state-reachability in non-recursive systems is EXPSPACE-complete [26, 29], for the moment, we do not know matching upper and lower bounds for RVASS.

Lemma 3. *The state-reachability problem for recursive (resp., non-recursive) vector addition systems without zero-test edges is EXPSPACE-hard, and in* 2*EXPTIME (resp., EXPSPACE).*

3.3 Encoding Recursively Parallel Programs as RVASSs

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 occurring 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, i.e., $\vec{n}_i(i) = 1$ 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 $\mathcal{A}_P = \langle Q, \delta \rangle$. We define $Q \overset{\text{def}}{=} \mathsf{Tasks} \cup \mathsf{Tasks}^3$, and define δ formally in Figure 7. 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. 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 $\langle T[\textbf{skip}]\,, t_0, t_f \rangle \in Q$ 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 eventually step past each **await** statement.

Notice that 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 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.

Lemma 4. *For all programs P without task-passing, procedures $p_0 \in \mathsf{Procs}$, and values $\ell_0, \ell \in \mathsf{Vals}$, ℓ 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 $\langle \ell, s, d \rangle$ is reachable from $\langle \ell_0, s_{p_0}, d_0 \rangle$ in $\mathcal{A}_P$.*

Our analysis algorithms in the following sections use Lemma 4 to compute state-reachability of a program P without task-passing by computing state-reachability on the corresponding RVASS $\mathcal{A}_P$.

$$\textsc{Additive}\quad \dfrac{q \overset{\vec{n}_1\vec{n}_2}{\hookrightarrow} q' \qquad \vec{n} \geq \vec{n}_1}{\langle q, \vec{n}\rangle\, c \xrightarrow{\text{rvas}} \langle q', \vec{n} \ominus \vec{n}_1 \oplus \vec{n}_2 \rangle\, c} \qquad \textsc{Call}\quad \dfrac{q \overset{q_1 q_2}{\hookrightarrow} q'}{\langle q, \vec{n}\rangle\, c \xrightarrow{\text{rvas}} \langle q_1, \mathbf{0}\rangle\, \langle q, \vec{n}\rangle\, c}$$

$$\textsc{Return}\quad \dfrac{q \overset{q_1 q_2}{\hookrightarrow} q'}{\langle q_2, \vec{n}_1\rangle\, \langle q, \vec{n}_2\rangle\, c \xrightarrow{\text{rvas}} \langle q', \vec{n}_1 \oplus \vec{n}_2 \rangle\, c} \qquad \textsc{Zero}\quad \dfrac{q \hookrightarrow q'}{\langle q, \mathbf{0}\rangle\, c \xrightarrow{\text{rvas}} \langle q', \mathbf{0}\rangle\, c}$$

Figure 6. 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, i.e., for all $q \in Q$, $\left|\delta_2 \cap (\{q\} \times Q^3)\right| \leq 1$. We denote by **0** the vector $\langle 0, 0, \ldots, 0\rangle$, and by $\oplus$ and $\ominus$ the usual vector addition and subtraction operators.

$$\dfrac{v_0 \in e(T) \qquad i = \mathsf{cn}(r, \langle v_0, s_p, d\rangle)}{T[\mathbf{post}\ r \leftarrow p\ e\ d] \overset{\mathbf{0}\vec{n}_i}{\hookrightarrow} T[\mathbf{skip}]} \qquad \dfrac{}{T[\mathbf{await}\ r] \hookrightarrow T[\mathbf{skip}]}$$

$$\dfrac{v_0 \in e(T) \qquad t_0 = \langle v_0, s_p, \mathtt{d}_{call}\rangle \qquad (\mathtt{l}\ \mathtt{:=}\ v_f) \in \mathsf{rvh}(t_f)}{T[\mathbf{call}\ \mathtt{l}\ \mathtt{:=}\ p\ e] \overset{t_0 t_f}{\hookrightarrow} T\left[\mathtt{l}\ \mathtt{:=}\ v_f\right]}$$

$$\dfrac{t_1 \xrightarrow[P]{\text{seq}} t_2}{t_1 \overset{\mathbf{00}}{\hookrightarrow} t_2} \qquad \dfrac{i = \mathsf{cn}(r, t_0) \qquad s \in \mathsf{rvh}(t_f)}{T[\mathbf{ewait}\ r] \overset{\vec{n}_i\mathbf{0}}{\hookrightarrow} \left\langle T[\mathbf{skip}], t_0, t_f \right\rangle \overset{t_0 t_f}{\hookrightarrow} T[s]}$$

$$\dfrac{i = \mathsf{cn}(r, t_0) \qquad s \in \mathsf{rvh}(t_f)}{T[\mathbf{await}\ r] \overset{\vec{n}_i\mathbf{0}}{\hookrightarrow} \left\langle T[\mathbf{skip}], t_0, t_f \right\rangle \overset{t_0 t_f}{\hookrightarrow} T[s\mathtt{;}\ \ \mathbf{await}\ r]}$$

Figure 7. The transitions of the RVASS $\mathcal{A}_P$ encoding the behavior of a finite-data recursively parallel program P.

In general, our algorithms compute sets of region valuation vectors

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

summarizing the execution of a procedure between an entry point t_0 and exit point t_f, where we write $\rightarrow^{\text{rvas}}_{\mathcal{A}_P} *$ to denote zero or more applications of $\rightarrow^{\text{rvas}}_{\mathcal{A}_P}$. 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.

4. Single-Wait Programs

Definition 2 (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.

4.1 Parallel Programming with Futures

The **future** annotation of Multilisp [17] has become a widely adopted parallel programming construct, included, for example, in X10 [6] and in Leijen et al. [25]'s task parallel library. Flanagan and Felleisen [12] 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,

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

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 every use of a variable assigned by a **future** statement is explicitly preceded by a **touch** statement. Semantically, the **future** 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 **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—i.e., 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 2. The Fibonacci function can be implemented as a parallel algorithm using futures as follows.

```
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
```

As opposed to the usual (naïve) sequential implementation operating in time $\mathcal{O}(n^2)$, this parallel implementation runs in time $\mathcal{O}(n)$.

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

$$\mathbf{future}\ x\ \mathtt{:=}\ p\ e \qquad \text{as} \qquad \mathbf{post}\ \mathtt{r}_x \leftarrow p\ e\ \vec{\mathtt{r}}\ \mathtt{d}_x$$

$$\mathbf{touch}\ x \qquad \text{as} \qquad \mathbf{ewait}\ \mathtt{r}_x$$

where $\mathtt{d}_x(v) \overset{\text{def}}{=} x\ \mathtt{:=}\ v$ 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.

4.2 Parallel Programming with Revisions

Burckhardt et al. [5]'s revisions model of concurrent programming proposes a mechanism analogous to (software) version control systems such as CVS and subversion, which promises to naturally and easily parallize sequential code in order to 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 their 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,

$$x\ \mathtt{:=}\ \mathbf{rfork}\ s \qquad \mathbf{join}\ x,$$

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—and all other processes created in this way. The assignment stores a *handle* to the newly-created revision in a *revision variable* 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 $m : (\mathsf{Vals} \times \mathsf{Vals} \times \mathsf{Vals}) \rightarrow \mathsf{Vals}$: when v_0, v_1 are, resp., 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 $m(v_0, v_1, v_2)$.

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

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

where p_s is a procedure declared as

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

and $\texttt{d}(\langle v_0, v_1 \rangle) \overset{\text{def}}{=} \texttt{l := } m(v_0\texttt{,l,}v_1)$ 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 $\vec{\texttt{r}}$ contains each $\texttt{r}_y$ for which the revision variable y is accessed in s.[2]

4.3 Programming with Asynchronous Procedures

Asynchronous programs [14, 19, 34] 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,

$$\textbf{async}\ p\ e \qquad \textbf{eventloop}$$

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—i.e., 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 non-deterministically.

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

$$\begin{array}{lll} \textbf{async}\ p\ e & \text{as} & \textbf{post}\ \texttt{r}_0 \leftarrow p'\ e\ \texttt{d} \\ \textbf{eventloop} & \text{as} & \textbf{while}\ \texttt{true}\ \textbf{do}\ \textbf{ewait}\ \texttt{r}_0. \end{array}$$

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 $\texttt{g}_0$ := $\star$
    var g := $\texttt{g}_0$
    $s$; return ($\texttt{g}_0$,g).
```

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

5. 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 non-recursive system) accounting for the left-over 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 "non-aliasing" case where the number of tasks stored in each region of a procedure frame is limited to one, the execution of **ewait** statements are deterministic. When the number of tasks stored in each region is not limited to one, non-determinism arises from the choice of which completed task to pick at each **ewait** statement (see the $\exists$WAIT rule of Figure 5). This added power makes the state-reachability problem at least as hard as state-reachability in vector addition systems—i.e., EXPSPACE-hard, though the precise complexity depends on the scope of pending tasks. After examining the PTIME-complete non-aliasing case, we examine two EXPSPACE-complete cases by restricting the scope of task handles, before moving to the general case.

5.1 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, $\texttt{r}_x$, contains at most one task handle. Thus the non-determinism arising (from choosing between tasks in a given region) in the $\exists$WAIT rule of Figure 3 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 child to parent.

Definition 3 (Non aliasing)**.** We say a region $r \in \mathsf{Regs}$ is *aliased* in a region valuation $m : \mathsf{Regs} \rightarrow \mathbb{M}[\mathsf{Tasks}]$ 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 *non-aliasing program* is a program in which no region is aliasing.

Note that the set of non-aliasing region valuations is finite when the number of program values is. The non-aliasing restriction thus allow us immediately to reduce the state-reachability problem for single-wait programs to reachability in a recursive finite-data sequential program. To compute state-reachability we consider a sequence $\mathcal{A}_0\mathcal{A}_1 \ldots$ of finite-state systems iteratively under-approximating 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

---

[2] Actually $\vec{\texttt{r}}$ must in general be chosen non-deterministically, as each revision handle may be joined either by the parent revision or its branch.

additive edge summarizing an $\mathbf{ewait}$ transition

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

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 $\mathcal{A}_{i-1}$, i.e., $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$. This $\mathcal{A}_0\mathcal{A}_1\ldots$ sequence is guaranteed to reach a fixed-point $\mathcal{A}_k$, since the set of non-aliasing 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 2. *The state-reachability problem for non-aliasing single-wait finite-value programs is PTIME-complete for a fixed number of regions, and EXPTIME-complete in the number of regions.*

5.2 Local-Scope Single-Wait Analysis

Definition 4 (Local scope)**.** A *local-scope program* is a program in which tasks only return with empty region valuations; i.e., for all reachable configurations $C[\langle t[\mathbf{return}\ e]\,, m\rangle]$ we have $m = m_\emptyset$.

To solve state-reachability in local-scope single-wait programs, we compute a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of non-recursive vector addition systems iteratively under-approximating 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 $\mathbf{post}$ transitions of P. At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing an $\mathbf{ewait}$ transition

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

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$; 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 $\mathcal{A}_0\mathcal{A}_1$ sequence is guaranteed to reach in a polynomial number of steps a fixed-point $\mathcal{A}_k$ whose reachable states are exactly those of $\mathcal{A}_P$. The entire procedure is EXPSPACE-complete, since each procedure-summarization reachability query is equivalent to computing state-reachability in vector addition systems.

Theorem 3. *The state-reachability problem for local-scope single-wait finite-value programs is EXPSPACE-complete.*

5.3 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 [34], 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 5 (Global scope)**.** A *global-scope programs* is a program in which the $\mathbf{ewait}$ (and $\mathbf{await}$) statements are used only in the initial procedure frame.

Since each non-initial 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 semi-linear set, described by the Parikh-image[3] of a context-free language. Following Ganty and Majumdar [14]'s 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 semi-linear set of tasks (recursively) posted between t_0 and t_f. Then, we use each $\mathcal{A}(t_0, t_f)$ as a component of a non-recursive 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 $t_0, t_f \in \mathsf{Tasks}$,

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

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 [14], this method constructs $\mathcal{A}'_P$ in polynomial time. Thus state-reachability in P is computed by state-reachability in the non-recursive vector addition system $\mathcal{A}'_P$, in exponential space. The complexity is asymptotically optimal since global-scope single-wait programs are powerful enough to capture state-reachability in vector addition systems.

Theorem 4. *The state-reachability problem for global-scope single-wait finite-value programs is EXPSPACE-complete.*

5.4 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.

Theorem 5. *The state-reachability problem for single-wait finite-value programs is EXPSPACE-hard, and in* 2*EXPTIME.*

Demri et al. [8]'s proof of membership in 2EXPTIME relies on a non-deterministically chosen reachability witness without materializing a practical algorithm for the search of said witness. Here we give a summarization-based algorithm.

To compute state-reachability we consider again a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of non-recursive vector addition systems successively under-approximating 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 $\mathbf{post}$ transitions of P. At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing an $\mathbf{ewait}$ transition

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

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$. Even though the set of possible added additive edges summarizing recursive transitions is infinite, with careful analysis we can show that this very simple algorithm terminates, 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 task t the set of vectors $\eta(t)$ needed to reach the target state in $\mathcal{A}_P$. Let us first recall some useful basic facts. Vector addition systems are monotonic w.r.t. the natural ordering on vectors of integers, i.e., if a transition is possible from a vector v, it is also possible from any u greater than v. The ordering on vectors of integers is a well quasi-ordering (WQO), i.e., in every sequence of vectors $v_0, v_1, \ldots$, there are two indices $i < j$ such that v_i is less or equal than v_j . Thus, every infinite set of vectors has a finite number of minimals. A set of vectors is upward closed if whenever it contains v it also contains all vectors greater than v. Such a set can be characterized by its minimals. Moreover,

---

[3] The Parikh-image of a word w over an alphabet Σ is the $|\Sigma|$-dimension vector of integers counting the number of occurrences of each symbol of Σ in w. The image of a language is the set of images of its elements.

the set of all predecessors in a vector addition system of an upward closed set of vectors is also upward closed; and therefore backward reachability analysis in these systems always terminates starting from an upward closed set [1, 11].

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 is *recursive* vector addition systems, we must solve several state-reachability queries on a sequence of vector addition systems with increasingly more transitions, which necessarily stabilizes. We elaborate below.

First, in order to reason backward about executions to the target state, consider the non-recursive system $\mathcal{A}'_i$ obtained by adding "return" transitions $t_f \overset{\mathbf{00}}{\hookrightarrow} T[s]$ from every procedure exit point $t_f = T_f[\mathbf{return}\ e]$ and procedure return point $T[\mathbf{ewait}\ e]$ occurring in P such that $s \in \mathsf{rvh}(t_f)$. These extra transitions in $\mathcal{A}'_i$ simulate a return from t_f to t, transferring all of the pending tasks from a frame at t_f to a frame at $T[s]$, without any contribution from the $T[s]$'s intra-procedural predecessor $T[\mathbf{ewait}\ e]$.

Then define a sequence of functions $\eta_0, \eta_1, \ldots : \mathsf{Tasks} \to \wp(\mathbb{N}^k)$, each η_i mapping each $t \in \mathsf{Tasks}$ to the (possibly empty, upward-closed) set of vectors $\eta_i(t)$ such that for any $\vec{n} \in \eta_i(t)$, a configuration $\langle t, \vec{n} \rangle$ is guaranteed to reach the target reachable state in $\mathcal{A}'_i$—and thus $\langle t, \vec{n} \rangle\ c$ is guaranteed to reach the target reachable state in $\mathcal{A}_P$ for any c; each η_i can be computed in by backward reachability in the non-recursive vector addition system as explained above. Since each $\mathcal{A}_i$ contains at least the transitions of $\mathcal{A}_{i-1}$, the η_i-sequence is non-decreasing w.r.t. set inclusion; i.e., more and more configurations can reach the target state; i.e., for all $t \in \mathsf{Tasks}$ we have $\eta_{i-1}(t) \subseteq \eta_i(t)$. 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 if a WQO), the η_i sequence must stabilize after a finite number of steps.

Furthermore, since any $\vec{n} \in \eta_i(t)$ is guaranteed to reach the target state, it suffices to consider only vectors $\vec{n}'$ bounded by the minimals of the upward-closed set $\eta_i(t)$. To see why, notice that if some $\vec{n} \in \eta_i(t)$ labels an edge between t_0 and t, then *every* configuration at t_0 is guaranteed to reach the target state, since this edge adds the vector guaranteed to reach the target from t. Additionally, any vector greater than a minimal of $\eta_i(t)$ is already guaranteed to be present in $\eta_i(t)$, since $\eta_i(t)$ is upward closed. Thus we need only consider edge-labels bounded by the decreasing $\eta_0\eta_1\ldots$ sequence, which shows that the $\mathcal{A}_0\mathcal{A}_1\ldots$ sequence stabilizes after a finite number of steps.

6. 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 6 (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.

6.1 Parallel Programming in Cilk

The Cilk parallel programming language [31] is an industrial-strength language with an accompanying runtime system which is used in a spectrum of environments, from modest multi-core computations to massively parallel computations with supercomputers. Similarly to futures (see Section 4.1), 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

$$\mathbf{inlet}\ p'\ (\mathbf{var}\ \mathtt{rv}\mathtt{:}\ T)\ s.$$

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 non-deterministically.

Example 3. The Fibonacci function can be implemented as a parallel algorithm using Cilk as follows.

```
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;
      return sum

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

As opposed to the usual (naïve) sequential implementation operating in time $\mathcal{O}(n^2)$, this parallel implementation runs in time $\mathcal{O}(n)$.

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

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

where $\mathtt{d}_{p'}(v) \overset{\text{def}}{=} s_{p'}[v/\mathtt{rv}]$ executes the top-level statement of the inlet p' with input parameter v.

6.2 Parallel Programming with Asynchronous Statements

The async/finish pair of constructs in X10 [6] 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,

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

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 4. The Fibonacci function can be implemented as a parallel algorithm using asynchronous statements as follows.

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

As opposed to the usual (naïve) sequential implementation operating in time $\mathcal{O}(n^2)$, this parallel implementation runs in time $\mathcal{O}(n)$.

Asynchronous statements are readily expressed with (non-deterministic) 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 $\mathtt{d}(\langle v_0, v_1 \rangle) \overset{\text{def}}{=} \textbf{assume } \mathtt{l} = v_0\mathtt{;}\ \mathtt{l} := v_1$ 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 4.3, 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 non-interfering tasks, by assuming either data-parallelism (i.e., disjoint accesses to data), or by assuming tasks are properly synchronized to ensure atomicity.

6.3 Structured Parallel Programming

So-called structured parallel constructs are becoming a standard parallel programming feature, adopted, for instance, in X10 [6] and in Leijen et al. [25]'s task parallel library. These constructs leverage normally sequential control structures to express parallelism. A typical syntactic instance of this is the parallel for-each loop:

$$\textbf{foreach } x \textbf{ in } e \textbf{ do } s$$

where x ranges over program variables, e over expressions, and s over statements. Semantically, the **foreach** 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 **foreach** statement then block 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 $\mathtt{r}_0$, we encode **foreach** x **in** e **do** s as

```
for $x$ in $e$ do post $\mathtt{r}_0$ ← $p_s$ ($x$,⋆) d;
await $\mathtt{r}_0$
```

and given that both x and l are free variables in s, p_s is a procedure declared as

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

and $\mathtt{d}(\langle v_0, v_1 \rangle) \overset{\text{def}}{=} \textbf{assume } \mathtt{l} = v_0\mathtt{;}\ \mathtt{l} := v_1$ models the update p performs from an initial (guessed) local valuation v_0.

7. 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 sub-case of local-scope programs, which in the multi-wait setting corresponds concurrency in the Cilk [31] language (modulo task interleaving), as well as structured parallel programming constructs such as the **foreach** parallel loop in X10 [6] and in Leijen et al. [25]'s task parallel library (see Section 6.3). The concurrent behavior of the asynchronous statements (Section 6.2) in X10 [6] does not satisfy the local-scope restriction, since **async** statements can include recursive procedure calls which are nested without interpolating **finish** statements. There computing state-reachability is equivalent to determining whether a particular vector is reachable in a non-recursive vector addition system—a decidable problem which is known to be EXPSPACE-hard, but for which the only known algorithms are non-primitive recursive. Since all multi-wait parallel languages we have encountered use only a single-region, we restrict our attention at present to single-region multi-wait programs.

7.1 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 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 requires determining the reachable program valuations between each pair of consecutive "synchronization points" (i.e., 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; i.e., when $T_1[\textbf{await } r]$ and $T_2[\textbf{await } r]$ 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 our reachability problem reveals it does not have the EXPSPACE-hard complexity of determining vector-reachability in general, due to the special structure of our reachability query. We notice that between two synchronization points t_1 and t_2 of p, execution proceeds in two phases. In the first, **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 can not be re-incremented again 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 $I(t_1, t_2)$ has a solution.

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 non-recursive vector addition systems iteratively under-approximating 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,

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

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

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

where $T_1[\mathbf{await}\ r]\,, T_2[\mathbf{await}\ r] \in \mathsf{Tasks}$ are consecutive synchronization points occurring P, and $\mathbf{0} \in \mathsf{sms}(T_1[\mathbf{skip}]\,, T_2[\mathbf{skip}]\,, \mathcal{A}_P)$. 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 $\mathcal{A}_0\mathcal{A}_1$ 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. The reachable states of $\mathcal{A}_k$ are precisely the same reachable states of $\mathcal{A}_P$.

Theorem 6. *The state-reachability problem for local-scope multi-wait single-region finite-value programs is NP-complete.*

7.2 Single-Region Multi-Wait Analysis

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, at a "synchronization point" $t_s \in \mathsf{Tasks}$ (note that $t_s = t_0$ if no **await** statements are encountered), and a **return** statement, at the exit point t_f. Since p can make recursive procedure calls between t_s and t_f, and each called procedure can again return pending tasks, the possible sets of pending tasks upon p's return at t_f is described by the Parikh-image[3] of a context-free language $L(t_0, t_f)$. It turns out we can describe this image as the set of vectors computed by a polynomially-sized vector addition system $\mathcal{A}^L(t_0, t_f)$ without recursion and zero-test edges [14]. We use thus computations of $\mathcal{A}^L(t_0, t_f)$ to summarize the set of possible region-valuations reached in an execution from t_0 to t_f. However, computing $\mathcal{A}^L(t_0, t_f)$ is not immediate, since between t_0 and the last-encountered synchronization point t_s, execution of the given procedure p may encounter **await** statements (necessarily so when $t_0 \neq t_s$). Since we use zero-test edges to express **await** statements, we also need to summarize execution between synchronization points (i.e., between the procedure entry point and among **await** statements) using only additive edges. To further complicate matters, each such summarization requires, in turn, the summaries $\mathcal{A}^L(t'_0, t'_f)$ computed for other procedures!

We break the circular dependence between procedure summaries and synchronization-point summaries by iteratively computing both. In particular, we compute a sequence $\mathcal{A}^L_0\mathcal{A}^L_1\ldots$ of procedure summary vector addition systems along with a sequence $\mathcal{A}_0\mathcal{A}_1\ldots$ of vector addition systems such that each $\mathcal{A}^L_i$, for $i > 0$, is computed using the transitions of $\mathcal{A}_{i-1}$, and $\mathcal{A}_i$, for $i \geq 0$ is computed using the procedure summaries of $\mathcal{A}^L_i$. Initially $\mathcal{A}^L_0$ contains only the pending-task sets reachable without taking **await** transitions, and $\mathcal{A}_0$ contains only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P, along with transitions to components $\mathcal{A}^L_0$. For $i \geq 0$, $\mathcal{A}_i$ contains transitions to and from the components $\mathcal{A}^L_i(t_0, t_f)$

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

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 unique initial and final states of $\mathcal{A}^L_i(t_0, t_f)$. (We assume each component $\mathcal{A}^L_i(t_0, t_f)$ has unique initial and final states, distinct from the states of other components. Additionally, we equip each $\mathcal{A}^L(t_0, t_f)$ with auxiliary state to carry the identity $T[\mathbf{skip}]$ of the invoking task to ensure the proper return of control when $\mathcal{A}^L(t_0, t_f)$ completes.)

At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing the execution between two synchronization points $T_1[\mathbf{await}\ r]$ and $T_2[\mathbf{await}\ r]$ occurring in P:

$$T_1[\mathbf{skip}] \xhookrightarrow{\mathbf{00}} T_2[\mathbf{skip}]$$

such that $T_2[\mathbf{skip}]$ is reachable in $\mathcal{A}_{i-1}$ from $T_1[\mathbf{skip}]$, i.e., $\mathbf{0} \in \mathsf{sms}(T_1[\mathbf{skip}]\,, T_2[\mathbf{skip}]\,, \mathcal{A}_{i-1})$. Note that when $T[\mathbf{await}\ r]$ is a synchronization point occurring in P, $T[\mathbf{skip}]$ refers to the program point immediately after the **await** statement. Since there are only polynomially-many such edges that can possibly be added, we are guaranteed to reach a fixed-point $\mathcal{A}_k$ of $\mathcal{A}_0\mathcal{A}_1\ldots$ in a polynomial number of steps. Furthermore, the reachable states of $\mathcal{A}_k$ are precisely the same reachable states of $\mathcal{A}_P$. However, computing $\mathbf{0} \in \mathsf{sms}(t_1, t_2, \mathcal{A}_{i-1})$ at each step is difficult due to the zero-test edge in the **await** statement immediately preceding t_2; this is computationally equivalent to computing reachability of a particular vector in non-recursive vector addition systems.

Theorem 7. *The state-reachability problem for multi-wait single-region finite-value programs is decidable.*

Since 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 over-approximate 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, practical algorithmic solutions are more probable (see Section 5.4).

8. Related Work

Formal modeling and verification of multi-threaded programs has been heavily studied, including but not limited to identifying decidable sub-classes [20], and effective over-approximate [13, 18] and under-approximate [9, 22] analyses.

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. Sen and Viswanathan [34]'s asynchronous programs, which falls out as a special case of our single-wait programs, is perhaps most similar to our work in this regard. Practical verification algorithms by combining iterative over- and under-approximation [19], and in-depth complexity analysis [14] of asynchronous programs have been studied.

Though decidability results of abstract parallel models have been reported [4, 10] (Bouajjani and Esparza [3] 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.

9. Conclusion

We have proposed a general model of recursively parallel programs which captures the concurrency constructs in a variety of popular 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 con-

State-Reachability in Recursively Parallel Programs

	result	complexity	language/feature
Task-Passing			
general	Thm. 1	undecidable	futures, revisions
Single-Wait			
non-aliasing	Thm. 2	PTIME	futures†, revisions†
local scope	Thm. 3	EXPSPACE	—
global scope	Thm. 4	EXPSPACE	asynchronous programs
general	Thm. 5	2EXPTIME	—
† For programs without task-passing.			
Multi-Wait (single region)			
local scope	Thm. 6	NP	Cilk
general	Thm. 7	decidable	async (X10)

Figure 8. Summary of results for computing state-reachability for finite-value recursively parallel programs.

current programming languages and program analyses. Figure 8 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, and the anonymous POPL reviewers.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS '96: Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society, 1996.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, Inc., 2006.

[3] A. Bouajjani and J. Esparza. Rewriting models of boolean programs. In *RTA '06: Proc. 17th International Conference on Term Rewriting and Applications*, volume 4098 of *LNCS*, pages 136–150. Springer, 2006.

[4] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR '05: Proc. 16th International Conference on Concurrency Theory*, volume 3653 of *LNCS*, pages 473–487. Springer, 2005.

[5] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA '10: Proc. 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 691–707. ACM, 2010.

[6] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005.

[7] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 159–169. ACM, 2008.

[8] S. Demri, M. Jurdzinski, O. Lachish, and R. Lazic. The covering and boundedness problems for branching vector addition systems. In *FSTTCS '09: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 181–192, 2009.

[9] J. Esparza and P. Ganty. Complexity of pattern-based verification for multithreaded programs. In *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 499–510. ACM, 2011.

[10] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL '00: Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11. ACM, 2000.

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

[12] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[13] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN '03: Proc. 10th International Workshop on Model Checking Software*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.

[14] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010. `http://arxiv.org/abs/1011.0551`.

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

[16] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proc. 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

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

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV '03: Proc. 15th International Conference on Computer Aided Verification*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.

[19] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 339–350. ACM, 2007.

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

[21] D. Kozen. Lower bounds for natural proof systems. In *FOCS '77: Proc. 18th Annual Symposium on Foundations of Computer Science*, pages 254–266. IEEE Computer Society, 1977.

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

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

[24] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA '09: Proc. 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–242. ACM, 2009.

[26] R. J. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

[27] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[28] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A programming model for deterministic task parallelism. In *MSPC '11: Proc. 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 7–12. ACM, 2011.

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

[30] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[31] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[32] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[33] C. Segulja and T. S. Abdelrahman. Synchronization-free and deterministic coarse-grain parallelism: Architectural support and programming model. In *FASPP '11: Proc. First International Workshop on Future Architectural Support for Parallel Programming*, 2011.

[34] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.

A. Proofs of Theorems

To begin with we introduce notation and simplifying assumptions in order to simplify the proof arguments in the following subsections.

Notation and Simplifying Assumptions

Words & Languages 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 $m \in \mathbb{M}[\Sigma]$ (equivalently, the vector $\vec{n} \in \mathbb{N}^{|\Sigma|}$) such that for each $a \in \Sigma$, $m(a)$ (resp., , $\vec{n}(a)$) 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 $\Pi(L_1) = \Pi(L_2)$.

Finite-State Automata 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 $\mathcal{A}(q_0, q_f) = \emptyset$.

Context-Free Grammars 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.

Pushdown Automata 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 $\mathcal{A}(q_0, q_f) = \emptyset$.

Vector Addition Systems A *vector addition system (VAS)* $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ 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 $\vec{n} \in \mathbb{N}^k$. Given initial and accepting states $q_0, q_f \in Q$, the language $\mathcal{A}(q_0, q_f)$ 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* (resp., the *configuration-reachability problem*) for vector addition systems is to decide, given a system $\mathcal{A}$ and states $q_0, q_f \in Q$ (resp., and a vector $\vec{n}_f \in \mathbb{N}^k$), whether $\mathcal{A}(q_0, q_f) \neq \emptyset$ (resp., whether $\vec{n}_f \in \mathcal{A}(q_0, q_f)$).

Turing Machines 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 $\mathcal{A}(q_0, q_f) = \emptyset$.

A.1 Proof of Theorem 1

Theorem 1. *The state-reachability problem for n-region finite-value task-passing parallel programs is undecidable for*

(a) non-recursive programs with $n > 1$, and
(b) recursive programs with $n > 0$.

We prove (a) 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 $\hookrightarrow = \{d_1, \ldots, d_j\}$, 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; i.e., $\langle q_f, a, \mathrm{L}, a, q_f \rangle \in \hookrightarrow$ for all $a \in \Sigma$. We define a task-passing program $P_{\mathcal{A}}$ with two regions $\mathtt{r}_L$ and $\mathtt{r}_R$, and one return-value handler d, along with an initial procedure given by

```
proc main ()
    var state: $Q$
    var sym: $\Sigma$
    var done: $\mathbb{B}$ = false

    while $\star$ do post $\mathtt{r}_R \leftarrow$ p $\star$ d;
    post $\mathtt{r}_R \leftarrow$ p $w(k)$ d;
    post $\mathtt{r}_R \leftarrow$ p $w(k-1)$ d;
    ...;
    post $\mathtt{r}_R \leftarrow$ p $w(2)$ d;

    state := $q_0$;
    sym := $w(1)$;

    while $\star$ do
        if $\star$ then $s_1$
        else if $\star$ then $s_2$
        ...
        else if $\star$ then $s_j$;

    // check: is state = $q_f$ reachable here?
    done := true;
    return
```

and an auxiliary non-recursive procedure p given by

```
proc p (var sym: $\Sigma$)
    return sym
```

where each transition $d_i \in \hookrightarrow$ gives rise to a corresponding statement s_i defined as follows. For right-moving transitions $d_i = q \xhookrightarrow{a/b,\mathrm{R}} q'$, we define s_i as

```
assume state = $q$;
assume sym = $a$;
post $\mathtt{r}_L \leftarrow p\ b$ d;
state := $q'$;
ewait $\mathtt{r}_R$ // overwrites sym
```

For left-moving transitions $d_i = q \xhookrightarrow{a/b,\mathrm{L}} q'$, we define s_i as

```
assume state = $q$;
assume sym = $a$;
post $\mathtt{r}_R \leftarrow p\ b$ d;
state := $q'$;
ewait $\mathtt{r}_L$ // overwrites sym
```

where the return-value handler $\mathtt{d}(a) \overset{\text{def}}{=}$ `sym := ` a assigns a to `sym`.

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 a valuation q_f to `state` at the end of the `main` procedure to know whether or not $\mathcal{A}$ has an accepting run.

Proposition A.1.I. *$\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if* $\mathtt{state} = q_f$ *and* $\mathtt{done} = \mathbf{true}$ *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 $\hookrightarrow\ = \{d_1, \ldots, d_j\}$, 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; i.e., $\langle q_f, a, \mathrm{L}, a, q_f \rangle \in \hookrightarrow$ for all $a \in \Sigma$. We define a single-region task-passing program $P_{\mathcal{A}}$ with a single return-value handler `d`, along with an initial procedure given by

```
proc main ()
    var q_cur, q_R: $Q$
    var sym_R: $\Sigma$
    var done: $\mathbb{B}$ = false

    while $\star$ do post r $\leftarrow$ p $\star$ d;
    post r $\leftarrow$ p $w(k)$ d;
    post r $\leftarrow$ p $w(k-1)$ d;
    ...;
    post r $\leftarrow$ p $w(1)$ d;

    ewait r;
    assume q_R = $q_0$;

    // check: is q_cur = $q_f$ reachable here?
    done := true;
    return
```

and an auxiliary recursive procedure `p` given by

```
proc p (var sym: $\Sigma$)
    var q_cur, q_init, q_R: $Q$
    var sym_R: $\Sigma$

    q_init := $\star$;
    q_cur := q_init;

    while $\star$ do
        if $\star$ then $s_1$
        else if $\star$ then $s_2$
        ...
        else if $\star$ then $s_j$
```

where each transition $d_i \in \delta$ gives rise to a corresponding statement s_i defined as follows. For the right-moving transitions $d_i = q \xhookrightarrow{a/b,\mathrm{R}} q'$, we define s_i as

```
assume q_cur = $q$;
assume sym = $a$;
sym := $b$;

ewait r;
// At this point q_R, q_cur, and sym_R
//   have been overwritten by the initial-
//   and current-state valuations, and the
//   symbol stored in the right-neighbor
//   who has just moved left.
assume q_R = $q'$;
post r $\leftarrow$ p sym_R d
```

where $\mathtt{d}(q, q', a)$ assigns q to `q_R`, q' to `q_cur`, and a to `sym_R`. Our program thus simulates right moves by awaiting a pending task representing the right neighbor of the current task. For left-moving transitions $d_i = q \xhookrightarrow{a/b,\mathrm{L}} q'$, we define s_i as

```
assume q_cur = $q$;
assume sym = $a$;
return (q_init, $q'$, $b$);
```

Our program thus simulates left moves by returning to the awaiting task, who promptly recreates its right-neighbor by posting a new task to replace it.

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 a valuation q_f to `q_cur` at the end of the `main` procedure to know whether or not $\mathcal{A}$ has an accepting run.

Proposition A.1.II. *$\mathcal{A}(q_0, q_f) \neq \emptyset$ if and only if* $\mathtt{q\_cur} = q_f$ *and* $\mathtt{done} = \mathbf{true}$ *is reachable in $P_{\mathcal{A}}$.*

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

Theorem A.1.I. *The state-reachability problem for single-region non-recursive finite-value task-passing parallel programs is PTIME-complete for fixed task-depth, and EXPTIME in the task-depth.*

Proof. Let P be a non-aliasing single-wait finite-value single-region non-recursive task-passing parallel program with finite sets of procedures Procs, values Vals, regions Regs, and return-value handlers Rets, and let $\ell \in \mathsf{Vals}$ be a target reachable value. Furthermore, we assume P is non-recursive, which implies there is a maximum task-depth $N \in \mathbb{N}$—i.e., 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 is only reachable in procedure frames where the current statement is s_f.

We construct a pushdown automaton $\mathcal{A}_P = \langle Q, \Sigma, \Gamma, \hookrightarrow \rangle$ along with initial and accepting states $q_0, q_f \in Q$. We define the states of $\mathcal{A}_P$ to be N-bounded sequences of tasks

$$Q \overset{\text{def}}{=} \mathsf{Tasks}^{\leq N}$$

In this way a state $t_0 t_1 \dots t_i \in Q$ 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:

Intra-task transitions For each intra-task transition $t_1 \rightarrow_P^{\text{seq}} t_2$ of Figure 2, we add the transition

$$t_1 \cdot \vec{t} \hookrightarrow t_2 \cdot \vec{t}.$$

POST For each statement $\mathbf{post}\ r \leftarrow p\ e\ d$ occurring in P, we add a transition which transfers control directly to procedure p,

$$T[\mathbf{post}\ r \leftarrow p\ e\ d] \hookrightarrow t \cdot T[\mathbf{skip}]\,,$$

where $t = \langle v, s_p, d \rangle$, for each $v \in e(T)$.

∃WAIT For each statement $\mathbf{ewait}\ r$ occurring in P, we add a transition which simply pops the pair $\langle v, d \rangle$ from the top of the pushdown stack, and applies the return-value handler,

$$T[\mathbf{ewait}\ r] \cdot \vec{t} \xhookrightarrow{\text{pop}\langle v,d \rangle} T[s] \cdot \vec{t},$$

where $s \in d(v)$.

RETURN For each statement $\mathbf{return}\ e$ occurring in P, we add a transition which pushes the return value and return-value handler for the current task onto the pushdown stack, to be later consumed by a subsequent **ewait** statement,

$$\langle \ell, S[\mathbf{return}\ e]\,, d \rangle \cdot t_0 \cdot \vec{t} \xhookrightarrow{\text{push}\langle v,d \rangle} t_0 \cdot \vec{t}.$$

where $v \in e(\ell)$

Finally, given an initial condition $\iota = \langle p_0, \ell_0 \rangle$ and target value ℓ_f of P, we let $q_0 = \langle \ell_0, s_{p_0}, d \rangle$, and $q_f = \langle \ell_f, s_f, d \rangle$, for some $d \in \mathsf{Rets}$. (See above for the definition of s_f.)

Proposition A.1.III. *$\mathcal{A}_P(q_0, q_f) \neq \emptyset$ if and only if ℓ is reachable in P from ι.*

As $|Q|$ is $\mathcal{O}((|\mathsf{Locs}| \cdot |\mathsf{Rets}|)^N)$ and $|\Gamma|$ is $\mathcal{O}(|\mathsf{Vals}| \cdot |\mathsf{Rets}|)$, the size of $\mathcal{A}_P$ is polynomial in P. 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 exponential in N. ☐

A.2 Proof of Theorem 2

Theorem 2. *The state-reachability problem for non-aliasing single-wait finite-value programs is PTIME-complete for a fixed number of regions, and EXPTIME-complete in the number of regions.*

Though our proof only handles local-scope programs, the extension to generally-scoped programs is possibly by allowing the values of the region-container variables `rg` below to be returned to waiting procedures.

Proof. Let P be a non-aliasing local-scope single-wait finite-value program with regions $\mathtt{r}_1, \dots, \mathtt{r}_n$. 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 := [ ⊥; ..; ⊥ ]
    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 $\mathbf{return}\ e$ into $\mathbf{return}\ (\mathtt{rg}, e)$, each statement $\mathbf{post}\ \mathtt{r}_i \leftarrow p\ e\ d$ into the assignment

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

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

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

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

Since regions do not alias, it is not hard to show that the state-reachability problem for the resulting sequential program P_{s} is equivalent to the state-reachability problem 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 ⋆ then
    rg := ⋆; rg' := ⋆; rv := ⋆;
    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 $\mathtt{l} = \ell$ reachable, if there is some reachable valuation with $\mathtt{l} = \ell$.)

Proposition A.2.I. *The value ℓ is reachable in P from ι if and only if ℓ 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 by n . Thus our state-reachability problem is PTIME-complete for fixed n since the state-reachability for sequential programs is [7, 32]. 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. ☐

A.3 Proof of Theorem 3

Theorem 3. *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)—i.e., 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 A.3.I. *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 sketch. To solve state-reachability in local-scope single-wait programs, we compute a sequence $\mathcal{A}_0 \mathcal{A}_1 \dots$ of non-recursive vector addition systems iteratively under-approximating 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

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

for some $t_0, t_f \in \mathsf{Tasks}$ such that $j = \mathsf{cn}(r, t_0)$, $s \in \mathsf{rvh}(t_f)$, and $\vec{n} \in \mathsf{sms}(t_0, t_f, \mathcal{A}_{i-1})$; 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 $\mathcal{A}_0\mathcal{A}_1$ 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 VASSs, we compute state-reachability in local-scope single-wait programs. $\square$

Lemma A.3.II. *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 (resp., awaited) for each addition (resp., 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 $\mathtt{d}_{const}(v) \overset{\text{def}}{=} \mathtt{l}$; i.e., $\mathtt{d}_{const}$ 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 $d_j = q \overset{\vec{n}_1\vec{n}_2}{\hookrightarrow} q'$, we define the statement s_j given by

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

Finally, the initial procedure is given by

```
proc main ()
    l := $q_0$;
    while $\star$ do
        if $\star$ then $s_1$
        else if $\star$ then $s_2$
        ...
        else if $\star$ then $s_{|\delta|}$.
```

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 $\mathcal{O}(|\mathcal{A}|^2)$. $\square$

A.4 Proof of Theorem 4

Theorem 4. *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)—i.e., 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 A.4.I. *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 sketch. Since each non-initial 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 semi-linear set, described by the Parikh-image of a context-free language. Following Ganty and Majumdar [14]'s 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 semi-linear set of tasks (recursively) posted between t_0 and t_f.

Proposition A.4.I ([14]). *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 non-recursive 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 $t_0, t_f \in \mathsf{Tasks}$,

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

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 $T[\textbf{skip}]$. 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 [14], this method constructs $\mathcal{A}'_P$ in polynomial time, reducing state-reachability in P to state-reachability in the VASS $\mathcal{A}'_P$ $\square$

Lemma A.4.II. *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 Lemma A.3.II from a given VASS $\mathcal{A}$ only uses the **ewait** statement in the initial procedure, $P_{\mathcal{A}}$ is also a global-scope program. $\square$

A.5 Proof of Theorem 5

Theorem 5. *The state-reachability problem for single-wait finite-value programs is EXPSPACE-hard, and in 2EXPTIME.*

To proceed we show an equivalence between the state-reachability problems of single-wait recursively parallel programs and recursive vector addition systems without zero-test edges—i.e., we show the problems are polynomial-time reducible to each other. EXPSPACE-hardness follows from that of non-recursive vector addition systems, and membership in 2EXPTIME follows from Demri et al. [8]'s result on branching vector addition systems (BVAS).

Lemma A.5.I. *The state-reachability problem for single-wait finite-value programs P over values Vals is reducible to the state-reachability problem for recursive vector addition systems in time $\mathcal{O}(|P| \cdot |\mathsf{Vals}|)$.*

Proof. The RVASS of $\mathcal{A}_P$ corresponding to a program P is given by Lemma 4 of Section 3.3; since P does not contain **await** statements, $\mathcal{A}_P$ does not contain zero-test edges. ☐

Lemma A.5.II. *The state-reachability problem for recursive vector addition systems $\mathcal{A}$ is reducible to the state-reachability problem for single-wait finite-value programs in time $\mathcal{O}(|\mathcal{A}|^2)$.*

Proof. Let $k \in \mathbb{N}$, and let $\mathcal{A} = \langle Q, \hookrightarrow \rangle$ be a RVASS over k-length vectors with additive transitions δ_1 and recursive transitions δ_2 (where $\hookrightarrow = \delta_1 \uplus \delta_2$), 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 procedure $\{p_q : q \in Q\}$ whose invocations will correspond to recursive transitions in $\mathcal{A}$, and a *dummy procedure* p_D which will be posted (resp., awaited) for each addition (resp., subtraction) performed in $\mathcal{A}$. Accordingly, the region-set $\mathsf{Regs} = \{\mathtt{r}_1, \dots, \mathtt{r}_k, \mathtt{r}_{call}\}$ of P contains a region $\mathtt{r}_i$ per vector component, and a *call region* $\mathtt{r}_{call}$. As the program's local variable `l` is used to store the control-state of $\mathcal{A}$, we set $\mathsf{Vals} = Q$. Finally, let $\mathsf{Rets} = \{\mathtt{d}_{const}\}$, where $\mathtt{d}_{const}(v) \overset{\mathrm{def}}{=} \ell$; i.e., $\mathtt{d}_{const}$ 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 $\hookrightarrow = \{d_1, \dots, d_n\}$. 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 l = $q$;
call l := $p_{q_2}$ $q_1$;
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 $d_i = q \xhookrightarrow{\vec{n}_1 \vec{n}_2} q'$, s_i is given by

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

Finally, the top-level statement for procedure p_q is

```
while  $\star$  do
    if 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 $i \in \{1, \dots, k\}$; 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 $\mathcal{O}(|\mathcal{A}|^2)$. ☐

A.6 Proof of Theorem 6

Theorem 6. *The state-reachability problem for local-scope multi-wait single-region finite-value programs is NP-complete.*

We show NP-hardness in Lemma A.6.I by a reduction from circuit satisfiability [27], and membership in NP in Lemma A.6.II by a procedure which solves a polynomial number of polynomial-sized integer linear programs.

Lemma A.6.I. *The circuit satisfiability problem [27] 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: $\mathbb{B}$, val: $\mathbb{B}$ }
```

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

```
proc set (var id: $W$, val: $\mathbb{B}$)
    var fst, snd: Wire

    if $\star$ then
        fst.id := id;
        fst.val := val;
        fst.active := true
    else
        snd.id := id;
        snd.val := val;
        snd.active := true;

    return (fst,snd)
```

which takes a value to be written and returns two output wires (one of which is written to), and a procedure for reading the value of a wire,

```
proc get (var id: $W$, fst, snd: Wire)
    var val: $\mathbb{B}$

    if $\star$ then
        assume fst.active and fst.id = id;
        val := fst.val;
        fst.active := false;
    else
        assume snd.active and snd.id = id;
        val := snd.val;
        snd.active := false;

    return (val,fst,snd).
```

which takes two wires `fst` and `snd`, reads a value from one of them, and returns the same (but mutated) wires, along with the value read. 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 $\mathtt{p}_g$ (var val: $\mathbb{B}$)
    var fst0, snd0, fst, snd: Wire
    var a, b, c: $\mathbb{B}$

    fst := fst0;
    snd := snd0;
    call (a,fst,snd) := get($w_1$,fst,snd);
    call (b,fst,snd) := get($w_2$,fst,snd);
    c := $f$(a,b);
    assume c = val;
    return (fst0,snd0,fst,snd).
```

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 $\mathtt{p}_g$, then waits until every task is consumed in some sequence,

```
proc init ()
    var fst, snd: Wire
    var val: $\mathbb{B}$
    var done: $\mathbb{B}$

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

    // input $h_1$
    val := $\star$;
    post r $\leftarrow$ set($w_{h_1,1}$,val) $\mathtt{d}_w$;
    post r $\leftarrow$ set($w_{h_1,2}$,val) $\mathtt{d}_w$;

    // input $h_2$
    val := $\star$;
    post r $\leftarrow$ set($w_{h_2,1}$,val) $\mathtt{d}_w$;
    post r $\leftarrow$ set($w_{h_2,2}$,val) $\mathtt{d}_w$;

    ...;

    // gate $g_1$
    val := $\star$;
    post r $\leftarrow$ $\mathtt{p}_{g_1}$(val) $\mathtt{d}_{rw}$;
    post r $\leftarrow$ set($w_{g_1,3}$,val) $\mathtt{d}_w$;
    post r $\leftarrow$ set($w_{g_1,4}$,val) $\mathtt{d}_w$;

    ...;

    await r;
    done := true,
```

where $w_{h_i,j}$ (resp., $w_{g_i,j}$) denotes the jth wire of input h_i (resp., gate g_i). The return handler $\mathtt{d}_w$(f,s) assigns `f` to `fst` and `s` to `snd`, and $\mathtt{d}_{rw}$(f0,s0,f,s) ensures `f0 = fst` and `s0 = snd`[4], and assigns `f` to `fst` and `s` to `snd`.

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 $\mathtt{p}_g$ 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 $\mathtt{p}_g$. 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

```
fst.id = $w_0$ and fst.val = true and done =
    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. ☐

Lemma A.6.II. *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 $\mathcal{O}(|P|^3 \cdot |\mathsf{Vals}|^3 \cdot |\mathsf{Rets}|)$-length series of integer linear programs, each of size $\mathcal{O}(|P|^5 \cdot |\mathsf{Vals}|^5 \cdot |\mathsf{Rets}|)$.*

Proof. Let P be a program with finite sets of procedures Procs, values Vals, and return-value handlers Rets, and let $\ell \in \mathsf{Vals}$ be a target reachable value from an initial condition $\iota = \langle p_0, \ell_0 \rangle$. We construct two sequences $\mathcal{A}^{\mathrm{s}}_1, \mathcal{A}^{\mathrm{s}}_2, \ldots$ and $\mathcal{A}^{\mathrm{t}}_1, \mathcal{A}^{\mathrm{t}}_2, \ldots$ of finite-state automata. Intuitively, each $\mathcal{A}^{\mathrm{s}}_i$ will be a *sync-point summary automaton*, characterizing pairs of program states reachable between two consecutive **await** statements; each $\mathcal{A}^{\mathrm{t}}_i$ will be a *task-summary automaton*, characterizing pairs of program states reachable between the entry and the exit of each task's procedure.

Let $Q \overset{\text{def}}{=} \mathsf{Tasks}$. We model task-posting by labeling the transitions of the automata by tasks, and define the alphabet $\Sigma \overset{\text{def}}{=} \mathsf{Tasks} \cup \{\varepsilon\}$. The initial task-summary automaton is $\mathcal{A}^{\mathrm{t}}_0 = \langle Q, \{\varepsilon\}, \emptyset \rangle$ with states Q, alphabet $\{\varepsilon\}$, and the empty set $\emptyset$ of transitions.

Construction of $\mathcal{A}^{\mathrm{s}}_i$ For $i > 0$, we define the i^{th} *sync-point summary automaton*, characterizing state-reachability between sync-point pairs, as

$$\mathcal{A}^{\mathrm{s}}_i = \left\langle Q \cup \bar{Q}, \Sigma, \delta^{\mathrm{s}}_i \right\rangle,$$

where the states Q and $\bar{Q} \overset{\text{def}}{=} \{\bar{q} : q \in Q\}$ correspond, resp., to control locations of the first (task-posting) and second (task-consuming) phases, and the transitions $\delta^{\mathrm{s}}_i = \delta^{+} \uplus \delta' \uplus \delta^{-}_i$ are partitioned into first-phase transitions $\delta^{+} \subseteq Q \times \Sigma \times Q$, phase-change transitions $\delta' = \{\langle q, \varepsilon, \bar{q} \rangle : q \in Q\}$, and second-phase transitions $\delta^{-}_i \subseteq \bar{Q} \times \Sigma \times \bar{Q}$.

The relation δ^{+} is given directly by the sequential and task-posting transitions of the input program. The relation δ^{-}_i contains a transition $\left\langle \bar{T}[\textbf{await } r], t_0, \bar{T}[s;\ \textbf{await } r] \right\rangle$ summarizing the computation of the task t_0 if and only if there exists $t_f \in \mathsf{Tasks}$ such that $\mathcal{A}^{\mathrm{t}}_{i-1}(t_0, t_f)$ is non-empty, and $s \in \mathsf{rvh}(t_f)$. In other words, $\langle \bar{q}, t, \bar{q}' \rangle$ summarizes the effect of consuming task t, based on $\mathcal{A}^{\mathrm{t}}_{i-1}$'s summarization of t, including the local-variable update due to its return-value handler. In this way, the possible behaviors between sync-points are computed using the thus-far computed (entire) behaviors of each posted task.

Note that not every word of $\mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ represents a valid computation between two consecutive sync points q_0 and q_f, since $\mathcal{A}^{\mathrm{s}}_i$ cannot ensure that each task posted in the first phase is consumed in the second. For $q_0, q_f \in Q$, we say a word $w_1 w_2 \in \mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ is *balanced* if and only if $\Pi(w_1) = \Pi(w_2)$ and there exists $q \in Q$ such that $w_1 \in \mathcal{A}^{\mathrm{s}}_i(q_0, q)$ and $w_2 \in \mathcal{A}^{\mathrm{s}}_i(\bar{q}, \bar{q}_f)$. We say $\mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ *has a balanced run* if some word of $\mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ is balanced. For each sync-point pair $\langle q_0, q_f \rangle$, we can decide whether $\mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ has a balanced run by integer linear programming. In particular, given $\mathcal{A}^{\mathrm{s}}_i$ and $\langle q_0, q_f \rangle$, we construct a integer linear program $\Phi^{\mathrm{s}}_i(q_0, q_f)$ which has a positive integer solution exactly when $\mathcal{A}^{\mathrm{s}}_i(q_0, \bar{q}_f)$ has a balanced run.

Construction of Φ^{s}_i Given the sync-point summary automaton $\mathcal{A}^{\mathrm{s}}_i$ and sync-point pair $q_0, q_f \in Q$, we construct an ILP, denoted $\Phi^{\mathrm{s}}_i(q_0, q_f)$. Fix (finite) enumerations $q_1 q_2 \ldots$, $a_1 a_2 \ldots$, and $d_1 d_2 \ldots$ of the states, symbols, and transitions, resp., of $\mathcal{A}^{\mathrm{s}}_i$; i.e., $Q = \{q_1, q_2, \ldots\}$, $\Sigma = \{a_1, a_2, \ldots\}$, and $\delta = \{d_1, d_2, \ldots\}$. Additionally, assume that $d_j = \langle q_j, \varepsilon, \bar{q}_j \rangle \in \delta'$ for each $q_j \in Q$. We define $\Phi^{\mathrm{s}}_i(q_0, q_f)$ as an integer linear program with $|\delta^{\mathrm{s}}_i|$ *transition occurrence variables*, one d_j for each transition $d_j \in \delta^{\mathrm{s}}_i$, and

---

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

$|\Sigma| - 1$ *task counter variables*, one a_j for each $a_j \in \Sigma \setminus \{\varepsilon\}$. Then $\Phi_i^{\mathrm{s}}(q_0, q_f)$ contains the following constraints: for each $q_k \in Q$,

$$\left(\mathrm{d}_k + \sum_{d_j \in \delta^+(q_k,\cdot,\cdot)} \mathrm{d}_j - \sum_{d_j \in \delta^+(\cdot,\cdot,q_k)} \mathrm{d}_j \right) = \begin{cases} 0 & \text{when } q_k \neq q_0 \\ 1 & \text{when } q_k = q_0 \end{cases}$$

ensures each state in the first phase is exited once per entry (except q_0, which is exited one extra time); for each $\bar{q}_k \in \bar{Q}$,

$$\left(\mathrm{d}_k + \sum_{d_j \in \delta_i^-(\cdot,\cdot,\bar{q}_k)} \mathrm{d}_j - \sum_{d_j \in \delta_i^-(\bar{q}_k,\cdot,\cdot)} \mathrm{d}_j \right) = \begin{cases} 0 & \text{when } q_k \neq q_f \\ 1 & \text{when } q_k = q_f \end{cases}$$

ensures each state in the second phase is exited once per entry (except q_f, which is entered one extra time);

$$\left(\sum_{d_j \in \delta'} \mathrm{d}_j \right) = 1$$

ensures a single inter-phase transition is taken; and for each $a_k \in \Sigma$,

$$\left(\sum_{d_j \in \delta^+(\cdot,a_k,\cdot)} \mathrm{d}_j \right) = \mathrm{a}_k = \left(\sum_{d_j \in \delta_i^-(\cdot,a_k,\cdot)} \mathrm{d}_j \right)$$

ensures that the number of occurrences of each a_k in the first phase is equal to the number of occurrence in the second phase. (Note that the a_j variables are not strictly necessary; they are added only for clarity.) Supposing $d_{j_1} d_{j_2} \ldots$ is a connected sequence of transitions through $\mathcal{A}_i^{\mathrm{s}}$, a corresponding solution to the given set of constraints would set the variables $\mathrm{d}_{j_1}, \mathrm{d}_{j_2}, \ldots$ to positive (non-zero) values corresponding to the number of times each transition is taken in $\mathcal{A}_i^{\mathrm{s}}$. However, supposing there are loops in $\mathcal{A}_i^{\mathrm{s}}$ 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 $\Phi_i^{\mathrm{s}}(q_0, q_f)$.

Proposition A.6.I. *$\mathcal{A}_i^{\mathrm{s}}(q_0, \bar{q}_f)$ has a balanced run if and only if $\Phi_i^{\mathrm{s}}(q_0, q_f)$ has a positive integer solution.*

Note that $|\Phi_i^{\mathrm{s}}|$ is bounded by $\mathcal{O}(|P|^5 \cdot |\mathsf{Vals}|^5 \cdot |\mathsf{Rets}|)$, since each of $\mathcal{O}(|Q|^2)$-many programs $\Phi_i^{\mathrm{s}}(q, q')$ contains $\mathcal{O}(|\delta_i^{\mathrm{s}}|) = \mathcal{O}(|Q|^2 \cdot |\Sigma|)$ variables and $\mathcal{O}(|Q| + |\Sigma|)$ constraints, where $\mathcal{O}(|Q|) = \mathcal{O}(|P| \cdot |\mathsf{Vals}|)$ and $\mathcal{O}(|\Sigma|) = \mathcal{O}(|P| \cdot |\mathsf{Vals}| \cdot |\mathsf{Rets}|)$.

Construction of $\mathcal{A}_i^{\mathrm{t}}$ For $i > 0$ we define the i^{th} task-summary automaton, characterizing state-reachability among synchronization points, as

$$\mathcal{A}_i^{\mathrm{t}} = \langle Q, \{\varepsilon\}, \delta_i^{\mathrm{t}} \rangle$$

such that $\langle q, \varepsilon, q' \rangle \in \delta_i^{\mathrm{t}}$ if and only if $\langle q, q' \rangle$ is a sync-point pair, and $\mathcal{A}_i^{\mathrm{s}}(q, q')$ has a balanced run.

Note that there are only finitely-many transitions which can be added over the entire $\mathcal{A}_i^{\mathrm{s}}$ and $\mathcal{A}_i^{\mathrm{t}}$ sequence. It follows that there exists a fixed-point $m \in \mathbb{N}$ of this sequence, and it is not hard to see that $\mathcal{A}_m^{\mathrm{s}}$ and $\mathcal{A}_m^{\mathrm{t}}$ capture every behavior of the input program P.

Proposition A.6.II. *A synchronization point q_f of the initial task is reachable from an initial control location q_0 if and only if $\mathcal{A}_m^{\mathrm{t}}(q_0, q_f)$ is non-empty.*

Though we consider here only state-reachability to a synchronization point contained in the initial task for simplicity, Proposition A.6.II can indeed be extended to arbitrary control locations of arbitrary tasks. As the set of possible added transitions is bounded by $\mathcal{O}(|Q|^2 \cdot |\Sigma|) = \mathcal{O}(|P|^3 \cdot |\mathsf{Vals}|^3 \cdot |\mathsf{Rets}|)$, our procedure is guaranteed to terminate in polynomial-time. $\square$

A.7 Proof of Theorem 7

Theorem 7. *The state-reachability problem for multi-wait finite-value programs is polynomial-time equivalent to the configuration-reachability problem for vector addition systems.*

We demonstrate this equivalence by a polynomial-time reduction in each direction. Though VASS configuration-reachability has been shown decidable [29], only non-primitive recursive algorithms are known; VASS state-reachability gives an EXPSPACE lower-bound.

Lemma A.7.I. *The state-reachability problem for multi-wait finite-value programs is reducible to the configuration-reachability problem for vector addition systems.*

Proof sketch. 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, at a "synchronization point" $t_s \in \mathsf{Tasks}$ (note that $t_s = t_0$ if no **await** statements are encountered), and a **return** statement, at the exit point t_f. Since p can make recursive procedure calls between t_s and t_f, and each called procedure can again return pending tasks, the possible sets of pending tasks upon p's return at t_f is described by the Parikh-image[3] of a context-free language $L(t_0, t_f)$. It turns out we can describe this image as the set of vectors computed by a polynomially-sized vector addition system $\mathcal{A}^L(t_0, t_f)$ without recursion and zero-test edges [14]. We use thus computations of $\mathcal{A}^L(t_0, t_f)$ to summarize the set of possible region-valuations reached in an execution from t_0 to t_f. However, computing $\mathcal{A}^L(t_0, t_f)$ is not immediate, since between t_0 and the last-encountered synchronization point t_s, execution of the given procedure p may encounter **await** statements (necessarily so when $t_0 \neq t_s$). Since we use zero-test edges to express **await** statements, we also need to summarize execution between synchronization points (i.e., between the procedure entry point and among **await** statements) using only additive edges. To further complicate matters, each such summarization requires, in turn, the summaries $\mathcal{A}^L(t_0', t_f')$ computed for other procedures!

We break the circular dependence between procedure summaries and synchronization-point summaries by iteratively computing both. In particular, we compute a sequence $\mathcal{A}_0^L \mathcal{A}_1^L \ldots$ of procedure summary vector addition systems along with a sequence $\mathcal{A}_0 \mathcal{A}_1 \ldots$ of vector addition systems such that each $\mathcal{A}_i^L$, for $i > 0$, is computed using the transitions of $\mathcal{A}_{i-1}$, and $\mathcal{A}_i$, for $i \geq 0$ is computed using the procedure summaries of $\mathcal{A}_i^L$. Initially $\mathcal{A}_0^L$ contains only the pending-task sets reachable without taking **await** transitions, and $\mathcal{A}_0$ contains only the transitions of $\mathcal{A}_P$ corresponding to intra-procedural and **post** transitions of P, along with transitions to components $\mathcal{A}_0^L$. For $i \geq 0$, $\mathcal{A}_i$ contains transitions to and from the components $\mathcal{A}_i^L(t_0, t_f)$

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

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 unique initial and final states of $\mathcal{A}_i^L(t_0, t_f)$. (We assume each component $\mathcal{A}_i^L(t_0, t_f)$ has unique initial and final states, distinct from the states of other components. Additionally, we equip each $\mathcal{A}^L(t_0, t_f)$ with auxiliary state to carry the identity $T[\mathbf{skip}]$ of the invoking task to ensure the proper return of control when $\mathcal{A}^L(t_0, t_f)$ completes.)

At each step $i > 0$, we add to $\mathcal{A}_i$ an additive edge summarizing the execution between two synchronization points $T_1[\mathbf{await}\ r]$ and $T_2[\mathbf{await}\ r]$ occurring in P:

$$T_1[\mathbf{skip}] \xhookrightarrow{\mathbf{00}} T_2[\mathbf{skip}]$$

such that $T_2[\textbf{skip}]$ is reachable in $\mathcal{A}_{i-1}$ from $T_1[\textbf{skip}]$, i.e., $\mathbf{0} \in \mathsf{sms}(T_1[\textbf{skip}], T_2[\textbf{skip}], \mathcal{A}_{i-1})$. Note that when $T[\textbf{await } r]$ is a synchronization point occurring in P, $T[\textbf{skip}]$ refers to the program point immediately after the **await** statement. Since there are only polynomially-many such edges that can possibly be added, we are guaranteed to reach a fixed-point $\mathcal{A}_k$ of $\mathcal{A}_0\mathcal{A}_1 \ldots$ in a polynomial number of steps. Furthermore, the reachable states of $\mathcal{A}_k$ are precisely the same reachable states of $\mathcal{A}_P$. However, computing $\mathbf{0} \in \mathsf{sms}(t_1, t_2, \mathcal{A}_{i-1})$ at each step is difficult due to the zero-test edge in the **await** statement immediately preceding t_2; this is computationally equivalent to computing configuration reachability in non-recursive vector addition systems. □

Lemma A.7.II. *The configuration-reachability problem for vector addition systems is reducible to the state-reachability problem for multi-wait finite-value programs.*

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 (resp., posted) corresponds to a subtraction (resp., 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

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

We assume, without loss of generality, that each transition has a non-zero decrement vector, i.e., $\vec{n}_1 \neq \mathbf{0}$ 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 $\mathtt{d}_{i,j}(v)$ be the return-value handler defined by

```
assume cur_tx = $i$;
assume cur_pos = $j$;
if cur_pos = $|\sigma_i|$ then
    assume $v$ = true;
    cur_tx := $\star$;
    cur_pos := 1
else
    assume $v$ = false;
    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$_{\vec{n}}$ ()
    for var idx := 1 to $k$ do
        for var cnt := 1 to $\vec{n}$(idx) do
            let tx = $\star$
            and pos = $\star$ in
            assume $\sigma_{\mathtt{tx}}$(pos) = idx;
            post r ← p$_{\mathtt{tx}}$ $\star$ d$_{\mathtt{tx,pos}}$.
```

which posts $\vec{n}(m)$ 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 $\vec{n}$. To perform the increment of transition $d_i \in \delta$ by vector $\vec{n}_2$, we declare the procedure $\mathtt{p}_i$, which non-deterministically calls $\mathtt{inc}_{\vec{n}_2}$, as

```
proc p$_i$ ()
    if $\star$ then
        call inc$_{\vec{n}_2}$ ();
        return true
    else
        return false.
```

Note that the Boolean return value is used by the attached return-value handler $\mathtt{d}_{i,j}$ (for some $j \in \{1, \ldots, |\sigma_i|\}$) to ensure that the increment is only performed once per transition d_i, by the last-consumed task in the $|\sigma_i|$-length sequence.

Finally, the initial procedure `main` simply adds tasks corresponding to the initial vector $\vec{n}_0$ to an initially-empty region container, then loops until every task has been consumed:

```
proc main ()
    var cur_tx :=  $\star$ ;
    var cur_pos = 1;
    var empty := false;
    call inc$_{\vec{n}_0}$ ();
    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. 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 A.7.I. *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 $\mathcal{A}$. □