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

HAL Id: hal-00639351
https://hal.science/hal-00639351v2
Submitted on 14 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
**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 2EXPSPACE 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 interweavings, 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 interweavings implicit in program syntax [23], 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].

---

*Partially supported by the project ANR-09-SEG1-016 Veridyc.
†Supported by a post-doctoral fellowship from the Fondation Sciences Mathématiques de Paris.
0Proofs to technical results are contained in the appendices.

---

1In 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].
already demonstrated effective algorithms for practically-occurring EXPSPACE-complete state-reachability problem instances based on co
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 parallelly executing tasks. Each task executes recursively 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 from 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, Exps a set of expressions, Rets a finite set of region identifiers, and 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 true and false, and Exps contains Vals and the (nullary) choice operator e. 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 ∈ N (resp., 1) region identifiers as n-region (resp., single-region) programs. A sequential program is a program without post, await, and await statements.

Each program P declares a sequence of procedures named p0...pn ∈ Procs, each p having single type-T parameter 1 and a top-level statement denoted sP; as statements are built inductively by composition with control-flow statements, sP describes the entire body of P. The set of program statements s is denoted Stmts. Intuitively, a post r ← p e 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 r’, and a return-value handler d. The await r statement blocks execution until some task whose handle is stored in region r completes, at which point its return-value handler is executed. Similarly, the await r statement blocks execution until all tasks whose handles are stored in region r complete, at which point all of their return-value handlers are executed, in some order. We refer to the call, post, return, await and await as inter-procedural statements, and the others as intra-procedural statements, and insist that return-value handlers are comprised only of intra-procedural statements. The assume e statement proceeds only when e evaluates to true—we use this statement in subsequent sections to block undesired executions in our encodings of other parallel programming models.

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

![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.](image-url)
as the return-value handler which simply denotes a transition of a task executing a 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 \( t = (\ell, S[s], d) \) is completed when its next-to-be-executed statement \( s \) is \( return \) \( c \), in which case we define \( rvh(t) \equiv \{ d(v) : v \in e(\ell) \} \) as the set of possible return-value handler statements for \( t \); \( rvh(t) \) is undefined when \( t \) is not completed.

Figure 2 and Figure 3 define the transition relation \( \rightarrow^{pp/p} \) of recursively parallel programs as a set of operational steps on configurations. The intra-procedural transitions of Figure 3 which implicitly include a transition \( C[(t_1, m_1)] \rightarrow^{pp/p} C[(t_2, m_1)] \) whenever \( t_1 \rightarrow^{\rho_1} t_2 \). 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 \( \exists\text{WAIT-T} \) 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 \( \forall\text{WAIT-}T \) and \( \forall\text{WAIT-DONE-T} \) 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 \( \text{post} \) and \( \text{await} \): supposing an unused region identifier \( r_{call} \), we translate each statement call \( 1 : p \in e \) into the sequence

\[
\text{post } r_{call} \leftarrow p \in e \\
\text{await } r_{call}
\]

where \( d_{call}(v) \equiv 1 : p \in e \) is the return-value handler which simply writes the entire return value \( v \) to the local variable \( 1 \), and \( \epsilon \) denotes an empty sequence of region identifiers.

A parallel execution of a program \( P \) (from \( c_0 \) to \( c_j \)) is a configuration sequence \( c_0; \ldots ; c_{j−1} \) where \( c_i \rightarrow^{pp/p} c_{i+1} \) for \( 0 \leq i < j \). An initial condition \( t = (p_0, \ell_0) \) is a procedure \( p_0 \in \text{Proc} \) along with a value \( \ell_0 \in \text{Vals} \). A configuration \( (\ell_0, s, d, m_0) \) is called \( (p_0, \ell_0) \)-initial when \( s \) is the top-level statement of \( p_0 \). A configuration \( c_j \) is called \( \ell_f \)-final when there exists a context \( C \) such that \( c_f = C[(t, m)] \) and \( 1(t) = \ell_f \). We say a valuation \( \ell \) is \( \text{reachable} \) in \( P \) from \( i \) when there exists an execution of \( P \) from some \( c_0 \) to \( c_f \), where \( c_0 \) is \( i \)-initial and \( c_f \) is \( \ell \)-final.

**Problem 1 (State-Reachability).** The state-reachability problem is to determine, given an initial condition \( i \) of a program \( P \) and a valuation \( \ell \), whether \( \ell \) is \( \text{reachable} \) in \( P \) from \( i \).

### 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 an \( \text{await} \) statement, suspending execution to wait for a subordinate task \( t \), becomes the currently-enabled task; when \( t \) completes, control returns to its waiting parent. At any moment only the tasks along one path \( p \) in the configuration tree have ever been enabled, and all but the last task in \( p \) are waiting for their child in \( p \) 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 \( \text{await} \) and \( \text{await} \) 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.

Figure 2 and Figure 3 define the sequential transition relation \( \rightarrow^{pp/s} \) of recursively parallel programs as a set of operational steps on configurations. The inter-procedural transitions of Figure 2 implicitly include a transition \( (t_1, m_1) \rightarrow^{pp/s} (t_2, m_2) \) whenever \( t_1 \rightarrow^{\rho_1} t_2 \). Interested here are the rules for \( \text{await} \) and \( \text{await} \). The \( \exists\text{WAIT-S} \) 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 \( \forall\text{WAIT-S} \) and \( \forall\text{WAIT-DONE-S} \) rules block the currently executing frame to obtain the results for every task in the given region, in a nondeterministically-chosen order. Finally, the \( \text{RETURN-S} \) applies a completed task’s return-value handler to update the parent frame’s
This capability makes the state-reachability problem undecidable—post (Task passing). We can then show that such tasks, it becomes possible to construct and manipulate unbounded essentially, when pending tasks can be passed to newly-created even for the very simple cases recursive programs with at least one

\[ T[\text{skip}; s] \xrightarrow{\text{skip}} T[s] \]

\[ T[\text{assume } e] \xrightarrow{\text{assume } e} T[\text{skip}] \]

\[ T[\text{if } e \text{ then } s_1 \text{ else } s_2] \xrightarrow{\text{if } e \text{ then } s_1 \text{ else } s_2} T[s_2] \]

\[ T[\text{while } e \text{ do } s] \xrightarrow{\text{while } e \text{ do } s} T[s; \text{ while } e \text{ do } s] \]

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 await 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.

The task-depth of a program \( P \) is the maximum length of a sequence \( p_1 \ldots p_t \) of procedures in \( P \) such that each \( p_i \) contains a statement post \( r \leftarrow p_{i+j} \in \tilde{d} \), for \( 0 < j < t \), and some \( r \in \text{Regs} \), \( e \in \text{Exprs} \), \( \tilde{d} \in \text{Regs}^* \), and \( d \in \text{Rets} \). Programs with unbounded task-depth are recursive, and are otherwise non-recursive.

**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 await 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 post \( r \leftarrow p \in d \). 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 \( m : \text{Regs} \rightarrow \mathbb{M}[\text{Tasks}] \) from regions to multisets of tasks, a frame \( \{t, m\} \) is a task \( t \in \text{Tasks} \) paired with a region valuation \( m \), and a configuration \( c \) is a sequence of frames representing a procedure activation stack. The transition relation \( \rightarrow^{\text{pp}} \) of Figures 2 and 5 implicitly include a transition \( \langle t_1, m \rangle \rightarrow^{\text{pp}} \langle t_2, m \rangle \) whenever \( t_1 \rightarrow_{\text{sim}} t_2 \). 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 \( \iota \) non-task-passing programs \( P \), the valuation \( \ell \) is reachable in \( P \) from \( \iota \) by a sequential execution

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

**Definition 1** (Task passing). A program which contains a statement post \( r \leftarrow p \in \tilde{d} \), such that \( |\tilde{d}| > 0 \) is called task-passing.
An execution of a RVASS $A$ (from $c_0$ to $c_j$) is a configuration sequence $c_0c_1 \ldots c_j$ where $c_i = \text{FRAME}_{i-1}$ for $0 \leq i < j$. A configuration $(q, \vec{n})$ is called $q_0$-initial when $q = q_0$ and $\vec{n} = \vec{0}$, and a configuration $q_j$ is called $q_j$-final when $q_j = (q_j, \vec{n})$ for some configuration $c$ and $\vec{n} \in \mathbb{N}^d$. We say a state $q_j$ is reachable in $A$ from $q_0$ when there exists an execution of $A$ from some $q_0$-initial configuration $c_0$ to some $q_j$-final configuration $c_j$. 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 2EXPSPACE. 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 2EXPSPACE (resp., EXPSPACE).

### 3.3 Encoding Recursively Parallel Programs as RVASSs

When the value set $\text{Vals}$ of a given program $P$ is taken to be finite, the set $\text{Tasks}$ 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 $m \in \text{Regs} \rightarrow \mathbb{N}[\text{Tasks}]$ by a vector $\vec{n} \in \mathbb{N}^d$ of counters, where $k = \vert \text{Regs} \times \text{Tasks} \vert$. To clarify the correspondence, we fix an enumeration $c_n : \text{Regs} \times \text{Tasks} \rightarrow \{1, \ldots, k\}$, and associate each region valuation $m$ with a vector $\vec{n}$ such that for all $r \in \text{Regs}$ and $t \in \text{Tasks}$, $m(r)(t) = \vec{n}(c_n(r,t))$. Let $n_i$ denote the unit vector of dimension $i$, i.e., $n_i(t) = 1$ and $n_i(j) = 0$ for $j \neq i$.

Given a finite-data recursively parallel program $P$ without task-passing, we associate a corresponding recursive vector addition system $A_P = (Q, \delta)$. We define $Q \equiv \text{Tasks} \cup \text{Tasks}^*$, and define $\delta$ formally in Figure [7]. Intra-procedural transitions translate directly to additive transitions. The calls 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 await statements are handled in two steps: first an additive transition decrements the counter corresponding to region-task pair $(r, t_0)$, and 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 $(T[\text{skip}], t_0, t_f) \in Q$ to connect the two transitions, in order to differentiate the intermediate steps of other await transitions.) The await statements are handled similarly, except the await statement must be repeated again upon the return. Finally, a zero-test transition allows $A_P$ to eventually step past each await statement.

Notice that ignoring intermediate states $(t_1, t_2, t_3) \in Q$, the frames $(l, \vec{n})$ of $A_P$ correspond directly to frames $(l, m)$ 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 $A_P$ and $P$.

**Lemma 4.** For all programs $P$ without task-passing, procedures $p_0 \in \text{Procs}$, and values $\ell_0, \ell \in \text{Vals}$, $\ell$ is reachable from $(\ell_0, p_0)$ in $P$ if and only if there exist $s \in \text{Stmts}$ and $d_0 \in \text{Regs}$ such that $(\ell, s, d)$ is reachable from $(\ell_0, s_0, d_0)$ in $A_P$.

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 $A_P$. 
The future construct leverages the procedural program structure computed value. Syntactically, futures add two statements, and Felleisen [12] provide a principled description of its semantics. in X10 [6] and in Leijen et al. [25]'s task parallel library. Flanagan modeled as single-wait programs.

In general, our algorithms compute sets of region valuation vectors $\mathbb{V}$ recursively parallel program $P$. Given an effective way to compute such a set, 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 $\mathbb{V}$ 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:

\[ \text{future } x := p e \quad \text{touch } x, \]

where $x$ ranges over program variables, $p \in \text{Procs}$, and $e \in \text{Exprs}$. 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.

\[
\begin{align*}
\text{proc fib (var n: } \mathbb{N}) &= \text{var x, y: } \mathbb{N} \\
&\text{if n < 2 then return 1} \\
&\text{else } \\
&\text{future x := fib (n-1)}; \\
&\text{future y := fib (n-2)}; \\
&\text{touch x; } \\
&\text{touch y; } \\
&\text{return x + y}
\end{align*}
\]

As opposed to the usual (naive) sequential implementation operating in time $O(n^2)$, this parallel implementation runs in time $O(n)$. The semantics of futures is readily expressed with task-passing programs using the post and ewait statements. Assuming a region identifier $r_x$, and return handler $d_x$ for each program variable $x$, we encode

\[
\begin{align*}
\text{future } x := p e & \quad \text{as } \text{post } r_x \leftarrow p e \exists d_x \\
\text{touch } x & \quad \text{as } \text{ewait } r_x
\end{align*}
\]

where $d_x(v) \equiv x := v$ simply assigns the return value $v$ to the variable $x$, and the vector $\exists d_x$ contains each $r_y$ such that the variable $y$ appears in $e$. 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,

\[
\begin{align*}
x := \text{rfork } s \\
\text{join } x
\end{align*}
\]
where $x$ ranges over program variables, and $s \in \text{Stmts}$. 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 $m : (\text{Vals} \times \text{Vals} \times \text{Vals}) \rightarrow \text{Vals}$: 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 await statements. Assuming a region identifier $r_x$ for each program variable $x$, and a programmer-supplied merge function $m$, we encode

\[
x := \text{rfork } s \text{ as } \text{post } r_x \leftarrow p_x \mid \mathcal{F} \mid d
\]

join $x$ as await $r_x$

where $p_x$ is a procedure declared as

\[
\text{proc } p_x \ (\text{var } l : T)
\begin{align*}
\text{var } l_0 & : = l \\
\text{return } (l_0, 1)
\end{align*}
\]

and $d((v_0, v_1)) \triangleq 1 := m(v_0, 1, v_1)$ updates the current local valuation based on the joined revision’s initial and final valuations $v_0, v_1 \in \text{Vals}$, and the joining revision’s current local valuation stored in $l$. The vector $\mathcal{F}$ contains each $r_x$ for which the revision variable $y$ is accessed in $s$.

### 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 shared 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. Syntaxically, asynchronous programs add two statements:

\[
\text{async } p e \quad \text{eventloop}
\]

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 await statements. Assuming a single region identifier $r_0$, we encode

\[
\text{async } p e \quad \text{as} \quad \text{post } r_0 \leftarrow p' e d
\]

\[
\text{eventloop} \quad \text{as} \quad \text{while true do} \text{await } r_0.
\]

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

\[
\begin{align*}
\text{proc } p' \ (\text{var } l : T) \\
\text{var } g_0 & := s \\
\text{var } g & := g_0 \\
s; \quad \text{return } (g_0, g).
\end{align*}
\]

Finally $d((r_0, v_1)) \triangleq 1 = v_0; 1 = v_1$ 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 $A_P$. To compute state-reachability in $P$ via procedure summarization, we must summarize the recursive transitions of $A_P$ 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 await 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 await statement (see the $\text{EWAIT}$ 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 $x$. 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, $r_x$, contains at most one task handle. Thus the non-determinism arising (from choosing between tasks in a given region) in the $\text{EWAIT}$ rule of Figure 5 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 $r \in \text{Regs}$ is aliasing in a region valuation $m : \text{Regs} \rightarrow \mathbb{M}[\text{Tasks}]$ when $m(r) > 1$.

We say $r$ is aliasing in a program $P$ if there exists a reachable configuration $C[[t, m]]$ of $P$ in which $r$ is aliasing 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 $A_0, A_1, \ldots$ of finite-state systems iteratively under-approximating the recursive system $A_P$ given from a single-wait program $P$. Initially, $A_0$ has only the transitions of $A_P$ corresponding to intra-procedural and post transitions of $P$. At each step $i > 0$, we add to $A_i$ an
The state-reachability problem for non-aliasing single-wait programs is PTIME-complete for a fixed number of regions, and EXPSPACE-complete in the number of regions.

5.2 Local-Scope Single-Wait Analysis

Definition 4 (Local scope). A local-scoped program is a program in which tasks only return with empty region valuations; i.e., for all reachable configurations \( C(t[\text{return } c], m) \) we have \( m = m_0 \).

To solve state-reachability in local-scoped single-wait programs, we compute a sequence \( A_0, A_1, \ldots \) of non-recursive vector addition systems iteratively under-approximating the recursive system \( A_P \) arising from a program \( P \). The initial system \( A_0 \) has only the transitions of \( A_P \) corresponding to intra-procedural and post transitions of \( P \). At each step \( i > 0 \), we add to \( A_i \) an additive edge summarizing an ewait transition

\[
T[\text{ewait } r] \xrightarrow{\vec{n}, \vec{n}'} T[s],
\]

for some \( t_0, t_f \in \text{Tasks} \) such that \( j = cn(r, t_0), s \in rvh(t_f), \) and \( \vec{n} \) is reachable at \( t_f \) from \( t_0 \) in \( A_{i-1} \), i.e., \( \vec{n} \in \text{sms}(t_0, t_f, A_{i-1}) \). This \( A_0, A_1, \ldots \) sequence is guaranteed to reach a fixed-point \( A_k \), since the set of non-aliasing region valuation systems, and thus the number of possibly added edges, is finite. Furthermore, as each \( A_i \) is finite-state, only finite-state reachability queries are needed to determine the reachable states of \( A_k \), which are precisely the same reachable states of \( A_P \). 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 EXPSPACE-complete in the number of regions.

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 \(^3\), 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-scoped program is a program in which the ewait (and await) statements are used only in the initial procedure frame.

Since each non-initial procedure \( p \) of a global-scoped 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 Ganry and Majumdar \(^4\)’s approach, for each \( t_0, t_f \in \text{Tasks} \) we construct a polynomial-sized vector addition system \( A(t_0, t_f) \)

characterizing this semi-linear set of tasks (recursively) posted between \( t_0 \) and \( t_f \). Then, we use each \( A(t_0, t_f) \) as a component of a non-recursive vector addition system \( A_P \) representing execution of the initial frame. In particular, \( A_P \) contains transitions to and from the component \( A(t_0, t_f) \) for each \( t_0, t_f \in \text{Tasks} \).

\[
T[\text{ewait } r] \xrightarrow{\vec{n}, \vec{n}'} \{q_0, T[\text{skip}]\} \quad (q_f, T[\text{skip}]) \xrightarrow{\vec{n}, \vec{n}'} T[s],
\]

for all \( r \in \text{Regs} \) such that \( j = cn(r, t_0), s \in rvh(t_f), \) and \( q_0 \) and \( q_f \) are the initial and final states of \( A(t_0, t_f) \). We assume each \( A(t_0, t_f) \) has unique initial and final states, distinct from the states of other components \( A(t_0', t_f') \). In order to transition to the correct state \( T[s] \) upon completion, \( A(t_0, t_f) \) carries an auxiliary state-component \( T[\text{skip}] \). In this way, for each task \( t \) posted to region \( r \) in an execution between \( t_0 \) and \( t_f \), the component \( A(t_0, t_f) \) does the incrementing of the \( cn(r', t') \)-component of the region-valuation vector. As each of the polynomially-many components \( A(t_0, t_f) \) are constructed in polynomial time \(^5\), this method constructs \( A_P \), in polynomial time. Thus state-reachability in \( P \) is computed by state-reachability in the non-recursive vector addition system \( A_P \), in exponential space. The complexity is asymptotically optimal since global-scoped single-wait programs are powerful enough to capture state-reachability in vector addition systems.

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

Theorem 4. The state-reachability problem for global-scoped 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 2EXPSPACE.

Demri et al. \(^8\)’s proof of membership in 2EXPSPACE 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 \( A_0, A_1, \ldots \) of non-recursive vector addition systems successively under-approximating the recursive system \( A_P \) of a single-wait program \( P \). Initially \( A_0 \) has only the transitions of \( A_P \) corresponding to intra-procedural and post transitions of \( P \). At each step \( i > 0 \), we add to \( A_i \) an additive edge summarizing an ewait transition

\[
T[\text{ewait } r] \xrightarrow{\vec{n}, \vec{n}'} T[s],
\]

for some \( t_0, t_f \in \text{Tasks} \) such that \( j = cn(r, t_0), s \in rvh(t_f), \) and \( \vec{n} \in \text{sms}(t_0, t_f, A_{i-1}) \). 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 \( A_P \). 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 \( A_P \). 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 \( v' \) 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 minimal. 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 minimal. Moreover,

\(^3\)The Parikh-image of a word \( w \) over an alphabet \( \Sigma \) is the \(|\Sigma|\)-dimension vector of integers counting the number of occurrences of each symbol of \( \Sigma \) 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 \( \mathcal{N} \).

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 \( \mathcal{A}' \) obtained by adding “return” transitions \( t_f \rightarrow T[s] \) from every procedure exit point \( t_f = T_f[\text{return } c] \) and procedure return point \( T[e\text{wait } c] \) occurring in \( P \) such that \( s \in rvh(t_f) \). These extra transitions in \( \mathcal{A}' \) 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 \( T[e\text{wait } c] \).

Then define a sequence of functions \( \eta_0, \eta_1, \ldots : \text{Tasks} \rightarrow \mathcal{P}([N]^k) \), each \( \eta_i \) mapping each \( t \in \text{Tasks} \) to the (possibly empty, upward-closed) set of vectors \( \eta_i(t) \) such that for any \( \vec{n} \in \eta_i(t) \), a configuration \((t, \vec{n}) \in \mathcal{A}' \) is guaranteed to reach the target reachable state in \( \mathcal{A}' \) — and thus \((t, \vec{n}) e \in \mathcal{A} \) is guaranteed to reach the target reachable state in \( \mathcal{A} \) for any \( e \); each \( \eta_i \) 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 \( \eta_i \)-sequence is non-decreasing w.r.t. set inclusion; i.e., more and more configurations can reach the target state; i.e., for all \( t \in \text{Tasks} \) 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 is a WQO), the \( \eta_i \) 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 minimal 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 \( t_0 \). 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 e\text{wait} 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 \( \text{[3]} \) 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 \( \text{[4]} \)), 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

\[
\text{spawn } p e p' \quad \text{sync},
\]

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

\[
\text{inlet } p' \quad (\text{var } r v: T) \quad s.
\]

Here \( s \) ranges over intra-procedural program statements containing two variables: \( r v \), corresponding to the value returned from a spawned procedure, and \( l \), corresponding to the local variable of the spawning procedure. Semantically, the \( \text{spawn} \) 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 \( \text{sync} \) 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.

\[
\begin{align*}
\text{proc fib (var n: N):}\quad & \text{var sum: N;}
\text{if } n < 2 \text{ then return 1 else spawn fib (n-1) summer;}
\text{spawn fib (n-2) summer; sync; return sum}
\end{align*}
\]

\[
\text{inlet summer (var i: N): sum := sum + i}
\]

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

The semantics of Cilk is ready expressed with recursively parallel programs using the post and await statements. Assuming a region identifier \( r_0 \), we encode

\[
\begin{align*}
\text{spawn } p e p' & \quad \text{as post } r_0 \leftarrow p e d_{p'}
\text{sync} & \quad \text{as await } r_0
\end{align*}
\]

where \( d_{p'}(v) \;\text{def}\; s_{p'}[v/r v] \) 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 \( \text{[6]} \) 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,

\[
\begin{align*}
\text{async } s & \quad \text{finish } s
\end{align*}
\]

where \( s \) ranges over program statements. Semantically, the \( \text{async} \) 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 \( \text{finish} \) 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.

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

As opposed to the usual (naive) sequential implementation operating in time \( O(n^2) \), this parallel implementation runs in time \( O(n) \).

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 \( r_1, \ldots, r_N \), we encode

```plaintext
async s as post \( r_i \) \( \leftarrow \) \( p_s \) \( \star \) \( d \)
finish s as await \( r_i \)
```

where \( i - 1 \) is number of enclosing `finish` statements, and \( p_s \) is a procedure declared as

```plaintext
proc p_s (var \( l: T \))
var \( l_0 \) := \( l \);
\( s \);
return \( (l_0, l) \)
```

and \( d((v_0, v_1)) \) is defined as `assume \( 1 = v_0; 1 := v_1 \)` 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, 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-loop:

```plaintext
foreach \( x \) in \( \mathbf{e} \) 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 \( r_0 \), we encode `foreach` `x` in `e` do `s` as

```plaintext
for \( x \) in \( e \) do post \( r_0 \) \( \leftarrow \) \( p_s \) \((x, s)\) \( d \);
await \( r_0 \)
```

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

```plaintext
proc p_s (var \( z: T; 1: T \))
var \( l_0 \) := \( l \);
\( s \);
return \( (l_0, l) \)
```

and \( d((v_0, v_1)) \) is defined as `assume \( 1 = v_0; 1 := v_1 \)` 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 \( A_P \) corresponding to a multi-wait program \( P \). As we have done for single-wait programs, we first examine the easier sub-case of local-scoped 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-scoping 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 \( p \in \text{Procs} \) between entry point \( t_0 \in \text{Tasks} \) and exit point \( t_f \in \text{Tasks} \) 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 \( A_P \), 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 \( 0 \) 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 \( t_1[\text{await } r_1] \) and \( t_2[\text{await } r_2] \) are consecutively-occurring synchronization points, we must determine whether \( (t_1[\text{skip}], 0) \) can reach \( (t_2[\text{skip}], 0) \).

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-scoping 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 \( A_0, A_1, \ldots \) of non-recursive vector addition systems iteratively under-approximating the recursive system \( A_P \). Initially \( A_0 \) has only the transitions of \( A_P \) corresponding to intra-procedural and `post` transitions of \( P \). At each step \( i > 0 \), we add to \( A_i \) 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,  

\[ T_{[\text{await } r]} \xrightarrow{\sigma_i^0} T[s; \text{ await } r], \]

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

\[ T_{1[\text{skip}]} \xrightarrow{00} T_{2[\text{skip}]}, \]

where \( T_{1[\text{await } r]}, T_{2[\text{await } r]} \in \text{Tasks} \) are consecutive synchronization points occurring \( P, \) and \( 0 \in \text{sms}(T_{1[\text{skip}]}, T_{2[\text{skip}]}, A_{i}). \) 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 \( A_{i} \) sequence is guaranteed to reach a fixed-point \( A_k \) 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 \( A_k \) are precisely the same reachable states of \( A_P. \)

**Theorem 6.** The state-reachability problem for local-scoped 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 \( p \in \text{Procs} \) between entry point \( t_0 \) in Tasks and exit point \( t_f \in \text{Tasks} \) is summarized by the tasks posted between the last-encountered await statement, at a “synchronization point” \( t_s \in \text{Tasks} \) (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\(^2\) 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 \( A^P(t_0, t_f) \) without recursion and zero-test edges\(^3\). We use thus computations of \( A^P(t_0, t_f) \) to summarize the set of possible region-valuations reached in an execution from \( t_0 \) to \( t_f. \) However, computing \( A^P(t_0, t_f) \) 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 \( A^P(t'_0, t'_f) \) 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 \( A^P_0, A^P_1, \ldots \) of procedure summary vector addition systems along with a sequence \( A_0, A_1, \ldots \) of vector addition systems such that each \( A^P_i, \) for \( i > 0, \) is computed using the transitions of \( A_{i-1}, \) and \( A_i, \) for \( i \geq 0 \) is computed using the procedure summaries of \( A^P_i. \) Initially \( A^P_0 \) contains only the pending-task sets reachable without taking await transitions, and \( A_0 \) contains only the transitions of \( A_P \) corresponding to intra-procedural and post transitions of \( P, \) along with transitions to components \( A^P_i. \) For \( i \geq 0, A_i \) contains transitions to and from the components \( A^P_i(t_0, t_f) \)

\[ T_{[\text{await } r]} \xrightarrow{\sigma_i^0} (q_0, T_{[\text{skip}]} \xrightarrow{00} T[s; \text{ await } r] \]

for each \( t_0, t_f \in \text{Tasks} \) such that \( j = cn(r, t_0), s \in rh(t_f), \) and \( q_0 \) and \( q_f \) are the unique initial and final states of \( A^P_i(t_0, t_f). \) (We assume each component \( A^P_i(t_0, t_f) \) has unique initial and final states, distinct from the states of other components. Additionally, we equip each \( A^P_i(t_0, t_f) \) with auxiliary state to carry the identity \( T_{[\text{skip}]} \) of the invoking task to ensure the proper return of control when \( A^P_i(t_0, t_f) \) completes.) At each step \( i > 0, \) we add to \( A_i \) an additive edge summarizing the execution between two synchronization points \( T_{1[\text{await } r]} \) and \( T_{2[\text{await } r]} \) occurring in \( P. \)

\[ T_{1[\text{skip}]} \xrightarrow{00} T_{2[\text{skip}]}, \]

such that \( T_{2[\text{skip}]} \) is reachable in \( A_{i-1}, \) from \( T_{1[\text{skip}]}, \) i.e., \( 0 \in \text{sms}(T_{1[\text{skip}]}, T_{2[\text{skip}]}, A_{i-1}). \) Note that when \( T_{[\text{await } r]} \) is a synchronization point occurring in \( P, T_{[\text{skip}]} \) 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 \( A_k \) of \( A_0, A_1, \ldots \) in a polynomial number of steps. Furthermore, the reachable states of \( A_k \) are precisely the same reachable states of \( A_P. \) However, computing \( 0 \in \text{sms}(t_1, t_2, A_{i-1}) \) 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 * do await \( r. \) 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\(^4\), and effective over-approximate\(^5\) and under-approximate\(^6\) 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\(^7\)’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\(^8\) and in-depth complexity analysis\(^9\) of asynchronous programs have been studied.

Though decidability results of abstract parallel models have been reported\(^10\) (Bouajjani and Esparza\(^11\) 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-

---

\( \sigma_i^0 \) denotes the empty behavior of \( i \)th step.

\( \text{sms} \) denotes the set of synchronization messages.

\( L(t_0, t_f) \) denotes the Parikh image of \( L(t_0, t_f). \)

\( \text{over-approximate, under-approximate} \) analyses.
State-Reachability in Recursively Parallel Programs

<table>
<thead>
<tr>
<th>Task-Passing</th>
<th>result</th>
<th>complexity</th>
<th>language/feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>Thm. 1</td>
<td>undecidable</td>
<td>futures, revisions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single-Wait</th>
<th>result</th>
<th>complexity</th>
<th>language/feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-aliasing</td>
<td>Thm. 2</td>
<td>PTIME</td>
<td>futures¹, revisions³</td>
</tr>
<tr>
<td>local scope</td>
<td>Thm. 3</td>
<td>EXPSPACE</td>
<td>—</td>
</tr>
<tr>
<td>global scope</td>
<td>Thm. 4</td>
<td>EXPSPACE</td>
<td>asynchronous programs</td>
</tr>
<tr>
<td>general</td>
<td>Thm. 5</td>
<td>2EXPTIME</td>
<td>—</td>
</tr>
</tbody>
</table>

† For programs without task-passing.

Multi-Wait (single region)

<table>
<thead>
<tr>
<th>result</th>
<th>complexity</th>
<th>language/feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>local scope</td>
<td>Thm. 6</td>
<td>NP</td>
</tr>
<tr>
<td>general</td>
<td>Thm. 7</td>
<td>decidable</td>
</tr>
</tbody>
</table>

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

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 \( \Sigma \)-word is a finite sequence \( w \in \Sigma^* \) of symbols from an alphabet \( \Sigma \); the symbol \( \varepsilon \) 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 \( m \in \mathbb{N}[\Sigma] \) (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 all words in \( L \). For a finite set \( Q \) of states and a finite set \( \Sigma \) of symbols, the language-emptiness problem for finite-state automata is to decide, given an automaton \( A \) and states \( q_0, q_f \in \Sigma \), \( f \subseteq \Sigma^* \), \( \Pi(f) \subseteq \Pi(L) \) for all words \( f \). The set of \( \Sigma \)-words \( w \) such that \( \Pi(w) \) is a set of dimension \( q_0 \vec{n}_f \) 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 \( A \) and states \( q_0, q_f \in \Sigma \) (resp., and a vector \( \vec{n}_f \in \mathbb{N}^k \)), whether \( A(q_0, q_f) \neq \emptyset \) (resp., whether \( \vec{n}_f \in A(q_0, q_f) \)).

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

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 \( A = (Q, \Sigma, \rightarrow) \) be a Turing machine with \( \rightarrow = \{d_1, \ldots, d_j\} \), and let \( q_0, q_f \in Q \). We assume, without loss of generality, that upon entering the accepting state \( q_f \), \( A \) performs a sequence of left-moves until reaching the end of the tape; i.e., \( (q_f, a, L, a, q_f) \in \rightarrow \) for all \( a \in \Sigma \). We define a task-passing program \( P_A \) with two regions \( r_L \) and \( r_R \), and one return-value handler \( d \), along with an initial procedure given by

```
proc main ()
var state: Q
var sym: \Sigma
var done: \emptyset
while * do post rR ← p * d;
post rR ← p w(k) d;
post rR ← p w(k-1) d;
...;
post rR ← p w(2) d;
state := q0;
sym := w(1);
while * do
  if * then s1
  else if * then s2
  ...;
  else if * then sj;
// check: is state = qf 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 \( \rightarrow \) gives rise to a corresponding statement \( s_i \) defined as follows. For right-moving transitions \( d_i = q_i \rightarrow q_j \), we define \( s_i \) as
assume state = q;
assume sym = a;
post r_L ← p b d;
state := q;
ewait r_R // overwrites sym

For left-moving transitions \( d_i = a/b \), we define \( s_i \) as

\[
\text{assume state = q;}
\]
\[
\text{assume sym = a;}
\]
\[
\text{post r_L ← p b d;}
\]
\[
\text{state := q;}
\]
\[
\text{ewait r_R // overwrites sym}
\]

where the return-value handler \( d(a) \) assigns \( a \) to \( \text{sym} \).

By connecting the configurations of \( (q_i, w_1, w_2) \) of \( A \) to the chain of tasks in region \( r_L \)—corresponding to the cells of \( w_1 \)—and the chain of tasks in region \( r_R \)—corresponding to the cells of \( w_2 \)—it is routine to show that \( P_4 \) faithfully simulates precisely the runs of \( A \). As we assume \( A \) moves to the left upon encountering the accepting state \( q_f \), we need only check reachability of a valuation \( q_f \) to state \( q \) at the end of the main procedure to know whether or not \( A \) has an accepting run.

**Proposition A.1.I**. \( A(q_0, q_f) \neq \emptyset \) if and only if \( \text{state} = q_f \) and \( \text{done} = \text{true} \) is reachable in \( P_4 \).

Thus state-reachability in \( P_4 \) solves language emptiness for \( A \).

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 \( A = (Q, \Sigma, \rightarrow) \) be a Turing machine with \( \rightarrow = \{d_1, \ldots, d_j\} \), and let \( q_0, q_f \in Q \). We assume, without loss of generality, that upon entering the accepting state \( q_f \), \( A \) performs a sequence of left-moves until reaching the end of the tape; i.e., \( (q_f, a, L, a, q_f) \in \rightarrow \) for all \( a \in \Sigma \). We define a single-region task-passing program \( P_4 \) 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: B = false

  while * do post r ← p * d;
  post r ← p w(k) d;
  post r ← p w(k - 1) d;
  ...;
  post r ← 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 * do
    if * then s_1
    else if * then s_2
    ...;
    else if * 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 \( d_i = a/b \), we define \( s_i \) as

\[
\text{assume q_R = q_f;}
\]
\[
\text{assume sym = a;}
\]
\[
\text{sym := b;}
\]
\[
\text{ewait r;}
\]
\[
\text{// At this point q_R, q_cur, and sym_R have been overwritten by the initial-}
\]
\[
\text{// and current-state valuations, and the}
\]
\[
\text{// symbol stored in the right-neighbor}
\]
\[
\text{// who has just moved left.}
\]
\[
\text{assume q_R = q_f;}
\]
\[
\text{post r ← p sym_R d}
\]

Our program thus simulates right moves by awaiting a pending task representing the right neighbor of the current task. For left-moving transitions \( d_i = a/b \), we define \( s_i \) as

\[
\text{assume q_cur = q_f;}
\]
\[
\text{assume sym = a;}
\]
\[
\text{sym := b;}
\]
\[
\text{ewait r;}
\]
\[
\text{// check: is q_cur = q_f reachable here?}
\]
\[
\text{done := true;
\]

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 \( (q_i, w_1, w_2) \) of \( A \) 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 \( P_4 \) faithfully simulates precisely the runs of \( A \). As we assume \( A \) moves to the left upon encountering the accepting state \( q_f \), we need only check reachability of a valuation \( q_f \) to \( q \) at the end of the main procedure to know whether or not \( A \) has an accepting run.

**Proposition A.1.II.** \( A(q_0, q_f) \neq \emptyset \) if and only if \( \text{q_cur} = q_f \) and \( \text{done} = \text{true} \) is reachable in \( P_4 \).

Thus state-reachability of \( P_4 \) solves language emptiness for \( A \).

**Theorem A.1.** 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 \( \text{Procs} \), values \( \text{Vals} \), regions \( \text{Regs} \), and return-value handlers \( \text{Ret} \), and let \( \ell \in \text{Vals} \) be a target reachable value. Furthermore, we assume \( \text{P} \) 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 \( p_0 p_1 \ldots p_k \in \text{Procs}^* \) such that each \( p_i \) contains a post to \( p_{i+1} \). Without loss of generality, suppose \( \ell \) is only reachable in procedure frames where the current statement is \( s_f \).

We construct a pushdown automaton \( A_P = (Q, \Sigma, \Gamma, \rightarrow) \) along with initial and accepting states \( q_0, q_f \in Q \). We define the states of \( A_P \) to be \( N \)-bounded sequences of tasks

\[
Q \triangleq \text{Tasks}^{\leq N}
\]
In this way a state \( t_0t_1 \ldots t_i \in Q \) represents a computation of \( P \) in which each \( t_j \) (where \( 0 < j \leq i \)) 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 \( \to \) on \( A_P \) as follows:

**Intra-task transitions** For each intra-task transition \( t_1 \to_P^\text{eq} t_2 \) of Figure 2, we add the transition

\[
  t_1 \cdot \tilde{t} \longrightarrow t_2 \cdot \tilde{t}.
\]

**POST** For each statement \( \text{post} \ r \leftarrow p \in d \) occurring in \( P \), we add a transition which transfers control directly to procedure \( p \),

\[
  T[\text{post} \ r \leftarrow p \in d] \longrightarrow t \cdot T[\text{skip}],
\]

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

**Ewait** For each statement \( \text{ewait} \ r \) occurring in \( P \), we add a transition which simply pops the pair \((v, d)\) from the top of the pushdown stack, and applies the return-value handler,

\[
  T[\text{ewait} \ r] \cdot \tilde{t} \longrightarrow \tilde{t} \cdot T[\text{skip}],
\]

where \( s \in d(v) \).

**Return** For each statement \( \text{return} \ e \) 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 \text{ewait} statement,

\[
  (\ell, S[\text{return} \ e], d) \cdot t_0 \cdot \tilde{t} \longrightarrow (\ell, t_0 \cdot \tilde{t}).
\]

Finally, given an initial condition \( \ell = (v_0, \ell_0) \) and target value \( \ell_f \) of \( P \), we let \( q_0 = (\ell_0, s_{p_0}, d) \), and \( q_f = (\ell_f, s_f, d) \), for some \( d \in \text{Rets} \). (See above for the definition of \( T_f \).)

**Proposition A.1.** \( A_P(q_0, q_f) \neq \emptyset \) if and only if \( \ell \) is reachable in \( P \) from \( \ell \).

As \( |Q| \) is \( O((|\text{Locs}| \cdot |\text{Rets}|)^N) \) and \( |\Gamma| \) is \( O(|\text{Vals}| \cdot |\text{Rets}|) \), the size of \( A_P \) 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.

Although our procedure only handles local-scope programs, the extension to globally-scoped programs is possible by allowing the values of the region-container variables \( \text{rg} \) below to be returned to waiting procedures.

**Proof.** Let \( P \) be a non-aliasing local-scope single-wait finite-value program with regions \( r_1, \ldots, r_n \). We define a sequential finite-value program \( P_r \) by a code-to-code translation of \( P \). We extend each procedure declaration \( \text{proc} \ p \) (\text{var} \ 1: \ T) \( s \) with additional procedure-local variables \( \text{rg}, \text{rg}' \), and \( \text{rv} \).

\[
  \text{proc} \ p \ (\text{var} \ 1: \ T) \quad \text{var} \ \text{rg}[n]: \ R := [\bot; \ldots; \bot] \\
  \text{var} \ \text{rg}'[n]: \ R \\
  \text{var} \ \text{rv}: \ T \\
  s
\]

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

\[
\{ \text{prc}: \text{Procs}, \text{arg}: \text{Vals}, \text{rh}: \text{Rets} \}.
\]

Note that \( R \) is a finite-type since \( \text{Procs}, \text{Vals}, \text{Rets} \) are finite sets. We translate each statement \( \text{return} \ e \) into \( \text{return} \ (\text{rg}, e) \), each statement \( \text{post} \ r \leftarrow p \in d \) into the assignment

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

and each statement \( \text{ewait} \ r_i \) into the statement

\[
\begin{align*}
  \text{assume} \ \text{rg}[i] &\neq \bot; \\
  \text{call} (\text{rg}', \text{rv}) &:= (\text{rg}[i].\text{prc}, \text{rg}[i].\text{arg}, 1 := \text{rg}[i].\text{rh}, \text{rg}[i] := \bot; \\
  \text{for} \ j := 1 \text{ to } n \text{ do} &\text{ if } \text{rg}'[j] \neq \bot \text{ then } \text{rg}[j] := \text{rg}'[j]
\end{align*}
\]

where we assume each \( d \in \text{Rets} \) is given by an expression in which \( \text{rv} \) is a free variable. Note that for local-scope programs, the \( \text{rg'} \) array will always be equal to \([\bot; \ldots; \bot]\) 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 \( P_r \) is equivalent to the state-reachability problem for \( P \). (Though technically we must check for reachability for a complete local valuation in \( P_r \), including \( 1, \text{rg}, \text{rg}' \), and \( \text{rv} \), we may assume without loss of generality reachability to certain values, by adding

\[
\begin{align*}
  \text{if} \ s \text{ then } \text{rg} := s; \text{rg}' := s; \text{rv} := s; \text{assume false}
\end{align*}
\]

between every statement of \( P_r \). Since the \text{assume false} statement cannot continue execution, this extra conditional statement has no effect on program behavior, besides making any valuation with \( 1 = \ell \) reachable, if there is some reachable valuation with \( 1 = \ell \).

**Proposition A.2.1.** The value \( \ell \) is reachable in \( P \) from \( i \) if and only if \( \ell \) is reachable in \( P_r \) from \( i \).

The size of \( P_r \) is polynomial in \( P \), while the number of variables in \( P_r \) increases by \( n \). Thus our state-reachability problem is PTIME-complete for fixed \( n \) since the state-reachability for sequential programs is \( \text{EXPTIME} \). When the number \( n \) of regions is not fixed, this state-reachability problem becomes \( \text{EXPTIME} \)-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.3.1.** 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 \( A_0, A_1, \ldots \) of non-recursive vector addition systems iteratively under-approximating the recursive system \( A_P \) arising from a program \( P \). The initial system \( A_0 \) has only the transitions of \( A_P \) corresponding to intra-procedural and post transitions of \( P \). At each step \( i > 0 \), we add to \( A_i \) an additive edge summarizing an ewait transition

\[
  T[\text{ewait} \ r] \longrightarrow [0] T[s]
\]
for some \( t_0, t_f \in \text{Tasks} \) such that \( j = cn(r, t_0), s \in \text{rvh}(t_f), \) and \( \vec{n} \in \text{sm}(t_0, t_f, A_{i-1}) \); since \( P \) is local-scope, every such \( \vec{n} \) must equal \( 0 \). Since the number of possibly added edges is polynomial in \( P \), the \( A_{0:} \)A sequence is guaranteed to reach in a polynomial number of steps a fixed-point \( A_f \) whose reachable states are exactly those of \( A_f \). 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.

**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 \( A = (Q, \rightarrow) \) be a \( k \)-dimension VASS, and let \( q_0, q_f \in Q \). We construct a single-wait program \( P_A \) with the initial condition \( \ell \) and target valuation \( \ell_f \) such that \( A(q_0, q_f) \neq \emptyset \) if and only if \( \ell_f \) is reachable in \( P_A \) from \( \ell \).

The program \( P_A \) contains only two procedures: an initial procedure \( \text{main}() \) and a dummy procedure \( p \) which will be posted (resp., awaited) for each addition (resp., subtraction) performed in \( A \). Accordingly, the region-set \( \text{Regs} = \{r_1, \ldots, r_k\} \) of \( P_A \) contains a region \( r_i \) per vector component. The program’s local variable \( l \) is used to store the control-state of \( A \), and we set \( \text{Vals} = Q \). Finally, let \( \text{Rets} = \{d_{\text{const}} \} \), where \( d_{\text{const}}(v) = \ell \); i.e., \( d_{\text{const}} \) is the return-value handler which ignores the return value, keeping the local valuation intact.

We simulate the transitions of \( A \) by awaiting a task from each region \( r_i \) once per decrement to the \( i \)-th vector component, and subsequently posting a task to each region \( r_i \) once per increment to the \( i \)-th vector component. Thus for each transition \( d_j = q_i \rightarrow q_j \), we define the statement \( s_j \) given by

\[
\begin{align*}
\text{assume } l &= q \\
\text{ewait } r_1; \ldots; \text{ewait } r_j; \ldots; \text{ewait } r_k; \\
&\quad \quad \text{\( \vec{n}_1 \) times} \\
&\quad \quad \text{post } r_1 &\leftarrow p \ast d_{\text{const}}; \ldots; \text{post } r_j &\leftarrow p \ast d_{\text{const}}; \\
&\quad \quad \text{\( \vec{n}_2 \) times} \\
&\quad \quad \text{\( \cdots \)} \\
&\quad \quad \text{post } r_k &\leftarrow p \ast d_{\text{const}}; \ldots; \text{post } r_k &\leftarrow p \ast d_{\text{const}}; \\
&\quad \quad \text{\( \vec{n}_3 \) times} \\
\end{align*}
\]

\( l : = q' \).

Finally, the initial procedure is given by

\[
\begin{align*}
\text{proc main() } \\
&\text{1 : = } q_0; \text{ while } * \text{ do } \\
&\quad \text{if } * \text{ then } s_1; \text{ else if } * \text{ then } s_2 \text{ \ldots } \\
&\quad \text{else if } * \text{ then } s_{|s|}. \\
\end{align*}
\]

Note the correspondence between configurations of \( A \) and \( P_A \). Each configuration \( (q, \vec{n}) \) of \( A \) maps directly to a configuration \( (q, s, d_{\text{const}}, m) \) of \( P_A \), where \( s \) is the loop state of the initial procedure, and \( |m(r_i)| = \vec{n}(i) \). Given this correspondence, it follows easily that the state \( q_f \) is reachable in \( A \) from \( q_0 \) if and only if the valuation \( \ell_f = q_f \) is reachable in \( P_A \) from \( \ell = (p_{\text{main}}, q_0) \). As there are \( O(|A|) \) statements in \( P_A \) per transition of \( A \), the size of \( P_A \) is \( O(|A|^2) \).

**A.4 Proof of 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’s approach, for each \( t_0, t_f \in \text{Tasks} \) we construct a polynomial-sized vector addition system \( A(t_0, t_f) \) characterizing this semi-linear set of tasks (recursively) posted between \( t_0 \) and \( t_f \).

**Proposition A.4.I** (14). For every pair \( t_0, t_f \in \text{Tasks}, \) region valuation \( m, \) and \( p \in \text{Procs}, \) there exists an execution of \( p \) from \( (t_0, m_0) \) to \( (t_f, m) \) if and only if there exists \( \vec{n} \in \mathbb{N}^k \) such that \( \vec{n} \in A(t_0, t_f) \), and \( m \) and \( \vec{n} \) represent the same Parikh-image.

We use each \( A(t_0, t_f) \) as a component of a non-recursive vector addition system \( A_P^P \) representing execution of the initial frame. In particular, \( A_P^P \) contains transitions to and from the component \( A(t_0, t_f) \) for each \( t_0, t_f \in \text{Tasks} \).

**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 \( P_A \) constructed in Lemma A.3.II from a given VASS \( A \) only uses the ewait statement in the initial procedure, \( P_A \) is also a global-scope program.

**A.5 Proof of Theorem 5**

The state-reachability problem for single-wait finite-value programs \( P \) over values \( \text{Vals} \) is reducible to the state-reachability problem for recursive vector addition systems in time \( O(|P| \cdot |\text{Vals}|) \).
Finally, the top-level statement for procedure $p_q$ is

$$\begin{align*}
\text{while} & \quad \ast \quad \text{do} \\
&\quad \text{if } l = q \quad \text{and} \quad * \quad \text{then return } \ast \\
&\quad \text{else if } * \quad \text{then } s_1 \\
&\quad \text{else if } * \quad \text{then } s_2 \\
&\quad \quad \quad \ldots \\
&\quad \text{else if } * \quad \text{then } s_n \\
&\quad \text{else skip.}
\end{align*}$$

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

### 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 gate $w_0$ true.

We construct a multi-wait single-region finite-value program $P_C$ as follows. Let $\text{wire}$ be the type defined as

$$\begin{align*}
\text{type Wire} & = \{ \text{id: } W, \text{ active: } \mathbb{B}, \text{ val: } \mathbb{B} \} \\
\text{var } \text{fst, snd: } \text{Wire}
\end{align*}$$

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

$$\begin{align*}
\text{proc set (var id: } W, \text{ val: } \mathbb{B}) & = \text{var } \text{fst, snd: } \text{Wire} \\
&\quad \text{if } * \text{ then } \\
&\quad \quad \text{fst.id := id;} \\
&\quad \quad \text{fst.val := val;} \\
&\quad \quad \text{fst.active := true} \\
&\quad \text{else} \\
&\quad \quad \text{snd.id := id;} \\
&\quad \quad \text{snd.val := val;} \\
&\quad \quad \text{snd.active := true;}
\end{align*}$$

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,

$$\begin{align*}
\text{proc get (var id: } W, \text{ fst, snd: } \text{Wire}) & = \text{var } \text{val: } \mathbb{B} \\
&\quad \text{if } * \text{ then } \\
&\quad \quad \text{assume } \text{fst.active and } \text{fst.id} = \text{id;} \\
&\quad \quad \text{val := } \text{fst.val;} \\
&\quad \quad \text{fst.active := false;} \\
&\quad \text{else} \\
&\quad \quad \text{assume } \text{snd.active and } \text{snd.id} = \text{id;} \\
&\quad \quad \text{val := } \text{snd.val;} \\
&\quad \quad \text{snd.active := false;}
\end{align*}$$

which takes two wires $\text{fst}$ and $\text{snd}$, 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.
We can block executions by allowing return handlers to be partial functions.

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

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 \( P_2 \) since every possible circuit evaluation order is considered. \( \square \)

**Lemma A.6.II.** The state-reachability problem for local-scope multi-wait single-region finite-value programs \( P \) over values \( \text{Vals} \) and return-value handlers \( \text{Rets} \) is reducible to solving a \( O(|\text{P}|^3 \cdot |\text{Vals}|^3 \cdot |\text{Rets}|) \)-length series of integer linear programs, each of size \( O(|\text{P}|^5 \cdot |\text{Vals}|^5 \cdot |\text{Rets}|) \).

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

Let \( Q \defeq \text{Tasks} \). We model task-posting by labeling the transitions of the automata by tasks, and define the alphabet \( \Sigma \defeq \text{Tasks} \cup \{\varepsilon\} \). The initial task-summary automaton is \( \mathcal{A}_0 = (Q, \{\varepsilon\}, \emptyset, \emptyset) \) with states \( Q \), alphabet \( \{\varepsilon\} \), and the empty set \( \emptyset \) of transitions.

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

\[
\mathcal{A}_i = (Q \cup \bar{Q}, \Sigma, \delta_i^+),
\]

where the states \( Q \) and \( \bar{Q} \defeq \{q : q \in Q\} \) correspond, resp., to control locations of the first (task-posting) and second (task-consuming) phases, and the transitions \( \delta_i^+ = \delta^+ \cup \delta^+ \delta_i^- \) are partitioned into first-phase transitions \( \delta^+ \subseteq Q \times \Sigma \times Q \), phase-change transitions \( \delta_i^- = \{q, \varepsilon, \bar{q} : q \in Q\} \), and second-phase transitions \( \delta_i^- \subseteq \bar{Q} \times \Sigma \times Q \).

The relation \( \delta^+ \) is given directly by the sequential and task-posting transitions of the input program. The relation \( \delta_i^- \) contains a transition \( \langle \bar{T}[\text{await } r], t_0, \bar{T}[s; \text{await } r] \rangle \) summarizing the computation of the task \( t_0 \) if and only if there exists \( t_f \in \text{Tasks} \) such that \( \mathcal{A}_{i-1}(t_0, t_f) \) is non-empty, and \( s \in \text{rvh}(t_f) \). In other words, \( \langle \bar{q}, t, \bar{q} \rangle \) summarizes the effect of consuming task \( t \), based on \( \mathcal{A}_{i-1} \)'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 \( \mathcal{A}_i(q_0, \bar{q}_f) \) represents a valid computation between two consecutive sync points \( q_0, q_f \), since \( A^i_0 \) cannot ensure that each task posted in the first phase is consumed in the second. For \( q_0, \bar{q}_f \in Q \), we say a word \( w_i, w_f \in \mathcal{A}_i(q_0, \bar{q}_f) \) is balanced if and only if \( \Pi(w_i) = \Pi(w_f) \) and there exists \( q \in Q \) such that \( w_i \in \mathcal{A}_0(q_0, q_f) \) and \( w_f \in \mathcal{A}_0(q, \bar{q}_f) \). We say \( \mathcal{A}_i(q_0, \bar{q}_f) \) has a balanced run if some word of \( \mathcal{A}_i(q_0, \bar{q}_f) \) is balanced. For each sync-point pair \( (q_0, q_f) \), we can decide whether \( \mathcal{A}_i(q_0, \bar{q}_f) \) has a balanced run by integer linear programming. In particular, given \( \mathcal{A}_i^1(q_0, \bar{q}_f) \), we construct a integer linear program \( \Phi_i(q_0, q_f) \) which has a positive integer solution exactly when \( \mathcal{A}_i(q_0, \bar{q}_f) \) has a balanced run.

**Construction of \( \Phi_i \)\(^\dagger\)** Given the sync-point summary automaton \( \mathcal{A}_i^1 \) and sync-point pair \( q_0, q_f \in Q \), we construct an ILP, denoted \( \Phi_i(q_0, q_f) \). Fix (finite) enumerations \( a, q_2, \ldots, a_q, 2, \ldots \) and \( d_1, d_2, \ldots \) of the states, symbols, and transitions, resp., of \( A^i_0 \); i.e., \( Q = \{q_1, q_2, \ldots\} \), \( \Sigma = \{a_1, a_2, \ldots\} \), and \( \delta = \{d_1, d_2, \ldots\} \). Additionally, assume that \( d_1 = (q_1, \varepsilon, \bar{q}_1) \in \delta_i^- \) for each \( q_1 \in Q \).

We define \( \Phi_i(q_0, q_f) \) as an integer linear program with \( |\delta| \) transition occurrence variables, one \( \delta_j \) for each transition \( \delta_j \in \delta_i^- \), and

4We can block executions by allowing return handlers to be partial functions.
For $\Phi(\{q_0, q_f\})$ contains the following constraints: for each $q_k \in Q$,

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

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

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

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

$$
\left( \sum_{d_j \in \delta^t} d_j \right) = 1
$$

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

$$
\left( \sum_{d_j \in \delta^+(\cdot, a_k \cdot)} d_j \right) = a_k = \left( \sum_{d_j \in \delta^-(\cdot, a_k \cdot)} d_j \right)
$$

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 $a_i$ variables are not strictly necessary; they are added only for clarity.) Supposing $d_1, d_2, \ldots$ is a connected sequence of transitions through $A^i$, a corresponding solution to the given set of constraints would set the variables $d_1, d_2, \ldots$ to positive (non-zero) values corresponding to the number of times each transition is taken in $A^i$. However, supposing there are loops in $A^i$ which are not connected to any of the selected transitions, the given constraints do not prohibit solutions which take each transition of these loops an arbitrary number of times. This is a standard issue with encoding automaton traces which can be addressed by adding a polynomial number of constraints to $P_i(q_0, q_f)$.

**Proposition A.6.I.** $A^i(q_0, q_f)$ has a balanced run if and only if $P_i(q_0, q_f)$ has a positive integer solution.

Note that $P_i$ is bounded by $O(|P|^3 \cdot |\text{Vals}|^3 \cdot |\text{Rets}|)$, since each of $O(|Q|^2)$ many programs $P_i(q_0, q_f)$ contains $O(|\delta_i|^3) = O(|Q|^2 \cdot |\Sigma|)$ variables and $O(|Q| + |\Sigma|)$ constraints, where $O(|Q|) = O(|P| \cdot |\text{Vals}|)$ and $O(|\Sigma|) = O(|P| \cdot |\text{Vals}| \cdot |\text{Rets}|)$.

**Construction of $A^i$**

For $i > 0$ we define the $i$th task-summary automaton, characterizing state-reachability among synchronization points, as

$$
A^i = \langle Q, \{e\}, \delta^i \rangle
$$

such that $(q, e, q') \in \delta^i$ if and only if $(q, q')$ is a sync-point pair, and $A^i(q_0, q_f)$ has a balanced run.

Note that there are only finitely-many transitions which can be added over the entire $A^i$ and $A^i$ sequence. It follows that there exists a fixed-point $m \in \mathbb{N}$ of this sequence, and it is not hard to see that $A^m$ and $A^m$ 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 $A^i(q_0, q_f)$ 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 $O(|Q|^2 \cdot |\Sigma|) = O(|P|^3 \cdot |\text{Vals}|^3 \cdot |\text{Rets}|)$, our procedure is guaranteed to terminate in polynomial-time.

**A.7 Proof of Theorem A.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.7.1.** 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 $p \in \text{Procs}$ between entry point $t_0 \in \text{Tasks}$ and exit point $t_f \in \text{Tasks}$ is summarized by the tasks posted between the last-encountered await statement, at a “synchronization point” $t_s \in \text{Tasks}$ (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 $t_0$ 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 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 $A^i(t_0, t_f)$ without recursion and zero-test edges [4]. We use thus computations of $A^i(t_0, t_f)$ to summarize the set of possible region-valuations reached in an execution from $t_0$ to $t_f$. However, computing $A^i(t_0, t_f)$ 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 $t_0 \neq t_f$). 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 $A^i(t'_0, t'_s)$ 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 $A^1(t_0, t_f), A^2(t_0, t_f), \ldots$ of procedure summary vector addition systems along with a sequence $A_0, A_1, \ldots$ of vector addition systems such that each $A^i$, for $i > 0$, is computed using the transitions of $A_{i-1}$, and $A_i$ for $i \geq 0$ is computed using the procedure summaries of $A^i$. Initially $A^1$ contains only the pending-task sets reachable without taking await transitions, and $A_0$ contains only the transitions of $A^i$ corresponding to intra-procedural and post transitions of $P$, along with transitions to components $A^2$. For $i \geq 0, A_i$ contains transitions to and from the components $A^i(t_0, t_f)$.

$$
T[\text{await } r] \xrightarrow{\pi_{0, T[\text{skip}]}^t} T[\text{skip}] \xrightarrow{\langle q_j, T[\text{skip}] \rangle^0 \rightarrow T[s; \text{ await } r]} \text{for each } t_0, t_f \in \text{Tasks such that } j = cn(r, t_0), s \in rh(t_f), \text{ and } q_0 \text{ and } q_f \text{ are the unique initial and final states of } A^i(t_0, t_f). \text{ (We equip each component } A^i(t_0, t_f) \text{ has unique initial and final states, distinct from the states of other components. Additionally, we equip each } A^i(t_0, t_f) \text{ with auxiliary state to carry the identity } T[\text{skip}] \text{ of the invoking task to ensure the proper return of control when } A^i(t_0, t_f) \text{ completes.)}
$$

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

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

Lemma A.7.11. The configuration-reachability problem for vector addition systems is reducible to the state-reachability problem for multi-wait finite-value programs.

Proof. Let \( A = \langle Q, \rightarrow \rangle \) be a \( k \)-dimension vector addition system with \( \sigma = \langle d_1, \ldots, d_k \rangle \), and let \( q_0, q_1 \in Q \). Instead of checking reachability of a vector \( \vec{n}_i \) from \( 0 \) in \( A \), we will instead solve an equally-hard problem of checking whether \( 0 \) is reachable from an initial vector \( \vec{n}_0 \). To do this we construct a multi-wait program \( P_A \) and a local valuation \( \ell \) which is reachable in \( P_A \) if and only if the configuration \( q_0 \) is reachable from \( q_\ell \) in \( A \).

We will construct \( P_A \) such that the number of pending tasks in a configuration is equal to the sum of vector components in a corresponding configuration of \( A \). We then simulate each step of \( A \), which subtracts \( \vec{n}_1 \in \mathbb{N}^k \) and adds \( \vec{n}_2 \in \mathbb{N}^k \), by consuming \( \sum \vec{n}_1(i) \) tasks and posting \( \sum \vec{n}_2(i) \) 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 \( \sigma_i \in \{ 1, \ldots, k \}^* \) of counter decrements as

\[
\sigma_i \triangleq \{11 \ldots 12 22 \ldots 2 k \ldots k \}
\]

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

\[
\begin{align*}
\text{assume } & \text{cur}_tx = i; \\
\text{if } & \text{cur}_pos = |\sigma_i| \text{ then} \\
& \text{assume } v = \text{true}; \\
& \text{cur}_tx := j; \\
& \text{cur}_pos := 1 \\
\text{else} & \\
& \text{assume } v = \text{false}; \\
& \text{cur}_pos := \text{cur}_pos + 1,
\end{align*}
\]

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

\[
\text{proc } \text{inc}_\ell ()
\]

\[
\text{for } \text{var } & \text{idx} := 1 \text{ to } k \text{ do} \\
& \text{for } \text{var } \text{cnt} := 1 \text{ to } \vec{n}(\text{idx}) \text{ do} \\
& \text{let } tx = * \\
& \text{and } \text{pos} = * \text{ in} \\
& \text{assume } \sigma_{tx}(\text{pos}) = \text{idx}; \\
& \text{post } r \leftarrow p_{tx} \times d_\text{ex.pos},
\]

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 \( \sigma_i \) (since \( \text{pos} \text{ is assigned } * \) of arbitrary transitions \( d_i \) (since \( \text{tx} \text{ is assigned } * \) such that \( \sigma_i(j) = m \)—this ensures that the subsequent consumption of a task with handler \( d_{i,j} \) 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 \( p_r \), which non-deterministically calls \( \text{inc}_\ell \), as

\[
\begin{align*}
\text{proc } & p_r () \\
& \text{if } * \text{ then} \\
& \text{call inc}_\ell \; (); \\
& \text{return } \text{true} \\
& \text{else} \\
& \text{return } \text{false}.
\end{align*}
\]

Note that the Boolean return value is used by the attached return-value handler \( d_{i,j} \) (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 \( \text{main} \) 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.

\[
\begin{align*}
\text{proc } & \text{main} () \\
& \text{var } \text{cur}_tx := *; \\
& \text{var } \text{cur}_pos = 1; \\
& \text{var } \text{empty} = \text{false}; \\
& \text{call inc}_\ell \; (); \\
& \text{await } r; \\
& \text{// check: is this point reachable?} \\
& \text{empty} = \text{true}; \\
& \text{return}.
\end{align*}
\]

Checking that \( P_A \) faithfully simulates \( A \) is easily done by noticing the correspondence between configurations \( q_{\ell} \) of \( A \) and configurations of \( P_A \) with \( \sum_i \vec{n}_i \) pending tasks. Since \( \text{empty} = \text{true} \) is only reachable when there are no pending tasks, reachability to \( \text{empty} = \text{true} \) implies \( 0 \) is reachable in \( A \). Furthermore, if \( 0 \) is reachable in \( A \), a run of \( P_A \) will eventually proceed past the \text{await} statement without pending tasks, setting \( \text{empty} = \text{true} \).

Proposition A.7.1. The configuration \( q_0 \) is reachable in \( A \) from \( q_{\ell} \) if and only if \( \text{empty} = \text{true} \) is reachable in \( P_A \).

Since the size of \( P_A \) is polynomial in \( A \), we have a polynomial-time reduction for deciding configuration-reachability in \( A \).