Efficient May Happen in Parallel Analysis for Async-Finish Parallelism

Jonathan K. Lee, Jens Palsberg, Rupak Majumdar, and Hong Hong

UCLA Computer Science Department, University of California, Los Angeles, USA

Abstract. For concurrent and parallel languages, the may-happen-in-parallel (MHP) decision problem asks, given two actions in the program, if there is an execution in which they can execute in parallel. Closely related, the MHP computation problem asks, given a program, which pairs of statements may happen in parallel. MHP analysis is the basis for many program analysis problems, such as data race detection and determinism checking, and researchers have devised MHP analyses for a variety of programming models.
We present algorithms for static MHP analysis of a storeless abstraction of X10-like languages that have async-finish parallelism and procedures. For a program of size n, our first algorithm solves the MHP decision problem in $O(n)$ time, via a reduction to constrained dynamic pushdown networks (CDPNs). Our second algorithm solves the MHP computation problem in $O(n \cdot \max(n, k))$ time, where k is a statically determined upper bound on the number of pairs that may happen in parallel. The second algorithm first runs a type-based analysis that produces a set of candidate pairs, and then it runs the decision procedure on each of those pairs. For programs without recursion, the type-based analysis is exact and gives an output-sensitive algorithm for the MHP computation problem, while for recursive programs, the type-based analysis may produce spurious pairs that the decision procedure will then remove. Our experiments on a large suite of X10 benchmarks suggest that our approach scales well. Our experiments also show that while k is $O(n^2)$ in the worst case, k is often $O(n)$ in practice.

1 Introduction

For concurrent and parallel languages, the may-happen-in-parallel (MHP) decision problem asks, given two actions in the program, if there is an execution in which they can execute in parallel. Closely related, the MHP computation problem asks, given a program, which pairs of statements may happen in parallel. MHP analyses are useful as a basis for tools such as data race detectors [6, 14] and determinism checkers.

In this paper we study MHP analysis of a storeless model of X10-like languages that have async-finish parallelism and procedures. In X10 [5], the *async* statement enables programs to create threads, while the $finish$ statement provides a form of synchronization. Specifically, a finish statement $finish\ s$ waits for termination of all *async* statement bodies started while executing s.

Researchers have studied static MHP analysis for a variety of storeless programming models. Roughly, there are three categories of decidability results.

First, consider models with threads and synchronization mechanisms such as rendezvous. In case there are no procedures, Taylor proved in his seminal paper [21] that the MHP decision problem is NP-complete for a set of tasks that each contains only straight-line code, even when the set of possible rendezvous is known. The decision problem becomes undecidable if, in addition, procedure calls are allowed [18]. The decision problem is decidable if restricted synchronization techniques, such as nested locks, are used [9], but the complexity is exponential. The async-finish concurrency constructs of X10-like languages are different from threads with synchronization idioms such as rendezvous and locks, so the intractability results above do not immediately apply; indeed, we demonstrate a linear-time algorithm for the decision problem.

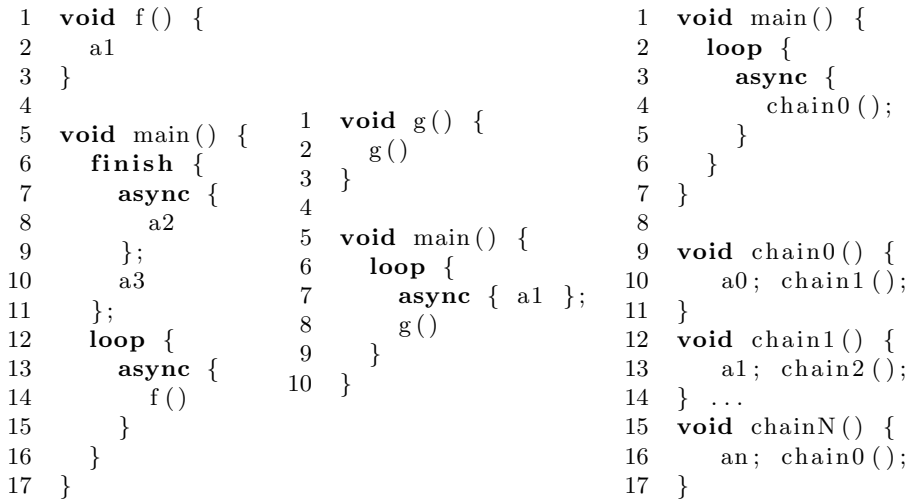
Second, consider models with syntactically specified synchronization, such as fork-join parallelism (e.g., Cilk). For fork-join parallelism, Seidl and Steffen [19] showed that the MHP decision problem is decidable in linear time. This result was extended by Lammich and Müller-Olm [10] in the presence of the *async* operator (called *spawn* in [10]) which can create new threads. Neither of these results immediately captures the *finish* construct of X10, in which an unbounded number of concurrently executing processes must synchronize. In the Seidl-Steffen paper, the fork-join construct ensures that there is at most a syntactically bounded number of processes executing and synchronizing in parallel. In the Lammich-Müller-Olm paper, spawned threads do not synchronize and synchronization is limited to an additional fork-join construct. Gawlitza et al. [8] made major progress and showed that MHP analysis is decidable for a model with nested locking and a join construct that has similarities with the *finish* construct in X10.

Finally, decidability results for MHP analysis have so far been mostly of theoretical interest. In particular, the decision procedures in [19, 19, 8] weren't applied to realistic benchmarks. Instead, most previous papers on practical MHP analysis present static analyses that give conservative, approximate answers to the MHP computation problem [7, 13, 15, 16, 12, 3, 1, 11]. The relationship between the approximate analyses and the theoretically optimal algorithms is unclear; if the theoretically optimal algorithms are also practically efficient, then that would make research into approximate analyses moot.

We study MHP analysis of Featherweight X10 [11], which is a core calculus for async-finish parallelism and procedures, and which is essentially a subset of X10. We give a store-less abstract semantics of Featherweight X10 and define the MHP decision problem and the MHP computation problem in terms of this semantics. The resulting MHP problems are all about control flow.

The challenge. For async-finish parallelism and procedures, is optimal MHP computation practical?

Our results. For Featherweight X10, we present two new algorithms for the MHP decision and computation problems, and we show that they scale well in practice.



```
1  void f() {
2    a1
3  }
4
5  void main() {
6    finish {
7      async {
8        a2
9      };
10     a3
11   };
12   loop {
13     async {
14       f()
15     }
16   }
17 }
```

```
1  void g() {
2    g()
3  }
4
5  void main() {
6    loop {
7      async { a1 };
8      g()
9    }
10 }
```

```
1  void main() {
2    loop {
3      async {
4        chain0();
5      }
6    }
7  }
8
9  void chain0() {
10   a0; chain1();
11 }
12 void chain1() {
13   a1; chain2();
14 } ...
15 void chainN() {
16   an; chain0();
17 }
```

Fig. 1. Three Featherweight X10 programs.

Our first algorithm solves the MHP decision problem in linear time, via a reduction from Featherweight X10 programs to constrained dynamic pushdown networks (CDPNs) [4]. We give a careful complexity analysis of a known decision procedure for CDPNs [4] for when it is applied to the CDPNs produced by our reduction.

Our second algorithm solves the MHP computation problem in $O(n{\cdot}\max(n,k))$ time, where k is a statically determined upper bound on the number of pairs that may happen in parallel. The second algorithm first runs a type-based analysis that produces a set of candidate pairs, and then it runs the decision procedure on each of those pairs. Following Lee and Palsberg [11], we recast the type analysis problem as a constraint solving problem that we can solve in $O(n \cdot \max(n,k))$ time. For programs without recursion, the type-based analysis is exact and gives an output-sensitive algorithm for the problem, while for recursive programs, the type-based analysis may produce spurious pairs that the decision procedure will then remove.

Our experiments on a large suite of X10 benchmarks suggest that our approach scales well. Our experiments also show that while k is $O(n^2)$ in the worst case, k is often $O(n)$ in practice. Thus, output-sensitivity is often crucial in getting algorithms to scale.

In summary, our results demonstrate two *tractable* MHP analyses for a *practical* parallel programming language.

In the following section we recall Featherweight X10 and give it an abstract semantics, and in Section 3 we define the MHP analysis problems. In Section 4 we present our type-based algorithm that produces a set of candidate pairs, in Section 5 we present our CDPN-based algorithm for the MHP decision problem, and in Section 6 we present our algorithm for solving the MHP computation

$$
\begin{array}{rrcl}
(\mathit{Statement}) & s & ::= & s\,;s \mid \mathit{loop}\ s \mid \mathit{async}\ s \mid \mathit{finish}\ s \mid a^l \mid \mathit{skip} \mid f() \\
(\mathit{Context}) & C & ::= & C\,;s \mid P\,;C \mid \mathit{async}\ C \mid \mathit{finish}\ C \mid \Box \\
(\mathit{ParStatement}) & P & ::= & P\,;P \mid \mathit{async}\ s \\
(\mathit{Redex}) & R & ::= & \mathit{skip}\,;s \mid P\,;\mathit{skip} \mid \mathit{loop}\ s \mid \mathit{async}\ \mathit{skip} \\
 & & & \mid \mathit{finish}\ \mathit{skip} \mid a^l \mid f()
\end{array}
$$

$$
\_[\_] : \mathit{Context} \times \mathit{Statement} \to \mathit{Statement}
$$

$$
(\Box)[s'] = s' \qquad (C\,;s)[s'] = (C[s'])\,;s \qquad (P\,;C)[s'] = P\,;(C[s'])
$$
$$
(\mathit{async}\ C)[s'] = (\mathit{async}\ C[s']) \qquad (\mathit{finish}\ C)[s'] = (\mathit{finish}\ C[s'])
$$

Fig. 2. Syntax of Featherweight X10

problem. Finally in Section 7 we present experimental results. We have omitted a large example and most of the proofs of correctness of our two algorithms; they are given in the appendices of the full version of the paper.

2 Featherweight X10

We now recall Featherweight X10 [11], and provide a store-less abstract semantics. In contrast to [11], we give a semantics based on evaluation contexts. Figure 1 shows three Featherweight X10 programs.

A program is a collection of procedures of the form

void $f()$ { s }

where f is a procedure name and s is the procedure body. We use $\mathsf{body}(f)$ to refer to the body of the procedure f. The procedure body is a statement generated by the grammar in Figure 2. We assume there is a procedure with the name *main*. The execution of a program begins by executing the body of *main*.

Syntax. Figure 2 gives the syntax of statements, contexts, parallel statements, and redexes, as well as a function for plugging a statement into a context. In the production for *Statement*, $s\,;s$ denotes statement sequence, *loop* s executes s zero, one, or more times, *async* s spawns off s in a separate thread, $finish\ s$ waits for termination of all *async* statement bodies started while executing s, a^l is a primitive statement with label l, *skip* is the empty statement, and $f()$ is a procedure call.

A context is a statement with a hole into which we can plug a statement. A parstatement is a statement in which multiple statements can execute in parallel. A redex is a statement that can execute at least one step of computation.

Featherweight X10 has no conditional statement; however, all the results in this paper can be extended easily to a conditional statement with nondeterministic branching.

The following theorem, proved by straightforward induction on s, characterizes statements in terms of contexts and redexes.

Theorem 1. (Statement Characterization) *For every statement s, either $s = skip$, or there exists a context C and a redex R such that $s = C[R]$.*

The characterization in Theorem 1 isn't necessarily unique. For example, if $s = (async\ a^5); (async\ skip)$, we can choose $C_1 = (async\ \Box); (async\ skip)$ and $R_1 = a^5$ and get $s = C_1[R_1]$, and we can choose $C_2 = (async\ a^5); \Box$ and $R_2 = async\ skip$ and get $s = C_2[R_2]$. The non-uniqueness reflects the nature of parallel computation: more than one statement can execute next, in some cases.

Abstract Semantics. We will define a small-step abstract store-less operational semantics. First we give some of the intuition behind the semantics by explaining how the semantics models the *finish* construct. Consider the statement:

$$(finish\ s_1); s_2 \tag{1}$$

Notice that the context $P\ ; C$ does not match (1) because $finish\ s_1$ is not a *ParStatement*. Thus, we cannot execute s_2. Rather, the only context that matches (1) is $C; s$. Thus, we will have to execute s_1 and if s_1 eventually becomes $skip$, then we will have rules that can bring us from $(finish\ skip); s_2$ to s_2.

We define a relation $\rightarrow \subseteq Redex \times Statement$:

$$skip\ ; s \rightarrow s \tag{2}$$
$$P\ ; skip \rightarrow P \tag{3}$$
$$loop\ s \rightarrow skip \tag{4}$$
$$loop\ s \rightarrow s\ ; loop\ s \tag{5}$$
$$async\ skip \rightarrow skip \tag{6}$$
$$finish\ skip \rightarrow skip \tag{7}$$
$$a^l\ \rightarrow skip \tag{8}$$
$$f()\ \rightarrow \mathsf{body}(f) \tag{9}$$

The program is fixed and implicit in the rules. Notice that for every redex R there exists s such that $R \rightarrow s$.

Intuitively, Rules (2)–Rule (3) say that *skip* is left unit for all statements and a right unit for *ParStatement*'s. Rules (4)–Rule (5) say that a loop executes its body zero or more times. Rules (6)–(7) say that *async* and *finish* have outplayed their roles when their body is *skip*. Rule (8) models primitive statements; in our store-less semantics, we don't record any effort. Rule (9) replaces a call to a procedure with the body of that procedure.

Next we define a relation $\longmapsto \subseteq Statement \times Statement$:

$$C[R] \longmapsto C[s] \quad \Longleftrightarrow \quad R \rightarrow s$$

We write $\longmapsto^*$ for the reflexive transitive closure of $\longmapsto$. The context $C\ ; s$ ensures that we can execute the first statement in a sequence, as usual. The contexts

$P\ ; C$ and $async\ C$ ensure that in a statement such as $(async\ s_1); (async\ s_2)$, we can execute either of s_1 or s_2 next. The context $finish\ C$ ensures that we can execute the body a finish statement.

3 The May-Happen-in-Parallel Problems

We now define the May Happen in Parallel decision and computation problems. We define:

$$\begin{aligned} \mathsf{CBE}(s, l_1, l_2) &= \exists C_1, C_2 : C_1 \neq C_2 \wedge s = C_1[a^{l_1}] = C_2[a^{l_2}] \\ \mathsf{CBE}(s) &= \{\ (l_1, l_2) \mid \mathsf{CBE}(s, l_1, l_2)\ \} \\ \mathsf{MHP}_{sem}(s) &= \bigcup_{s' : s \longmapsto^* s'} \mathsf{CBE}(s') \end{aligned}$$

Intuitively, $\mathsf{CBE}(s, l_1, l_2)$ holds if statements labeled l_1 and l_2 can both execute at s. We use the subscript sem in MHP_{sem} to emphasize that the definition is semantics-based.

For example, if $s = (async\ a^5); a^6$, we can choose $C_1 = (async\ \Box); a^6$ and $R_1 = a^5$ and get $s = C_1[R_1]$, and we can choose $C_2 = (async\ a^5); \Box$ and $R_2 = a^6$ and get $s = C_2[R_2]$. We conclude $\mathsf{CBE}(s, 5, 6)$ and $(5, 6) \in \mathsf{CBE}(s)$.

We define the MHP decision problem as follows.

May Happen in Parallel (decision problem)
Instance: (s, l_1, l_2) where s is a statement and l_1, l_2 are labels.
Problem: $(l_1, l_2) \in \mathsf{MHP}_{sem}(s)$?

Equivalently, we can phrase the decision problem as: does there exist s' such that $s \longmapsto^* s'$ and $\mathsf{CBE}(s', l_1, l_2)$?

We define the MHP computation problem as follows.

May Happen in Parallel (computation problem)
Input: a statement s.
Output: $\mathsf{MHP}_{sem}(s)$.

4 A Type System for producing Candidate Pairs

We now present a type system that gives a conservative solution to the MHP computation problem.

Type Rules. We define

$$\begin{aligned} symcross\ &:\ Set \times Set \to PairSet \\ symcross(S_1, S_2) &= (S_1 \times S_2) \cup (S_2 \times S_1) \end{aligned}$$

We use *symcross* to help produce a symmetric set of pairs of labels.

$$\frac{B \vdash s_1 : M_1, O_1, L_1 \qquad B \vdash s_2 : M_2, O_2, L_2}{B \vdash s_1 \ ; s_2 : M_1 \cup M_2 \cup symcross(O_1, L_2), O_1 \cup O_2, L_1 \cup L_2} \tag{10}$$

$$\frac{B \vdash s : M, O, L}{B \vdash loop\ s : M \cup symcross(O, L), O, L} \tag{11}$$

$$\frac{B \vdash s : M, O, L}{B \vdash async\ s : M, L, L} \tag{12}$$

$$\frac{B \vdash s : M, O, L}{B \vdash finish\ s : M, \emptyset, L} \tag{13}$$

$$B \vdash a^l : \emptyset, \emptyset, \{l\} \tag{14}$$

$$B \vdash skip : \emptyset, \emptyset, \emptyset \tag{15}$$

$$B \vdash f() : M, O, L \qquad (\text{if } B(f) = (M, O, L)) \tag{16}$$

$$\frac{B \vdash s_i : M_i, O_i, L_i \qquad B(f_i) = (M_i, O_i, L_i) \qquad i \in 1..n}{\vdash \textbf{void}\ f_1()\{\ s_1\ \}\ \ldots\ \textbf{void}\ f_n()\{\ s_n\ \} : B} \tag{17}$$

Fig. 3. Type rules.

We will use judgments of the forms $B \vdash s : M, O, L$ and $\vdash p : B$. Here, s is a statement, p is a program, M is a set of label pairs, O and L are sets of labels, and B is a type environment that maps procedure names to triples of the form (M, O, L). The meaning of $B \vdash s : M, O, L$ is that in type environment B, (1) the statement s has MHP information M, (2) while s is executing statements with labels in L will be executed, and (3) when s terminates, statements with labels in O may still be executing. The meaning of $\vdash p : B$ is that the program p has procedures that can be described by B. Figure 3 shows the eight rules for deriving such judgments.

Notice that if a derivation of $\vdash p : B$ contains the judgment $B \vdash s : M, O, L$, then $O \subseteq L$.

Let us now explain the eight rules in Figure 3. Rule (10) says that we can combine information for s_1 and information for s_2 into information for $s_1; s_2$ mainly by set union and also by adding the term $symcross(O_1, L_2)$ to the set of pairs. The role of $symcross(O_1, L_2)$ is to capture that the statements (with labels in O_1) that may still be executing when s_1 terminates may happen in parallel with the statements (with labels in L_2) that will be executed by s_2. Rule (11) has the term $symcross(O_1, L_2)$ as part of the set of pairs because the loop body may happen in parallel with itself. Rule (12) says that the body of *async* may still be executing when the *async* statement itself terminates. Note here that the second piece of derived information is written as L rather than $O \cup L$ because, as noted above, $O \subseteq L$. Rule (13) says that no statements in the body of $finish$ will still be executing when the $finish$ statement terminates. Rule (14) states that just the statement a^l will execute. Rule (15) states no labeled statements will execute. Rule (16) states that B contains all the information we need about a procedure. Rule (17) says that if B correctly describes every procedure, then it correctly describes the entire program.

Example. As an example, let us show a type derivation for the first program in Figure 1. Let

$$B = [\ \ f \mapsto (\emptyset, \emptyset, \{1\}),\ \ \ main \mapsto (\{(1,1),(2,3)\}, \{1\}, \{1,2,3\})\ ]$$

From Rule (17) we have that to show that the entire program has type B, we must derive the following two judgments:

$$B \vdash \mathsf{body}(f) : \emptyset, \emptyset, \{1\} \tag{18}$$

$$B \vdash \mathsf{body}(main) : \{(1,1),(2,3)\}, \{1\}, \{1,2,3\}) \tag{19}$$

Let us consider those judgments in turn.

We have that $\mathsf{body}(f) = a^1$ so Rule (14) gives us the judgment (18).

We have that $\mathsf{body}(main) = s_1; s_2$ where

$$s_1 = finish\ \{\ async\ \{\ a^2\ \};\ a^3\ \}$$
$$s_2 = loop\ \{\ async\ \{\ f()\ \}\ \}$$

From Rules (13), (10), (12), (14), we can produce this derivation:

$$\dfrac{\dfrac{\dfrac{B \vdash a^2 : \emptyset, \emptyset, \{2\}}{B \vdash async\ \{\ a^2\ \} : \emptyset, \{2\}, \{2\}} \qquad B \vdash a^3 : \emptyset, \emptyset, \{3\}}{B \vdash async\ \{\ a^2\ \};\ a^3 : \{(2,3)\}, \{2\}, \{2,3\}}}{B \vdash s_1 : \{(2,3)\}, \emptyset, \{2,3\}}$$

From Rules (11), (12), (16), we can produce this derivation:

$$\dfrac{\dfrac{B \vdash f() : \emptyset, \emptyset, \{1\}}{B \vdash async\ \{\ f()\ \} : \emptyset, \{1\}, \{1\}}}{B \vdash s_2 : \{(1,1)\}, \{1\}, \{1\}}$$

Finally, we can use Rule (10) to produce the judgment (19).

Properties. The following four theorems are standard and have straightforward proofs.

Theorem 2. (Existence of Typing) *For all B, there exists M, O, L such that $B \vdash s : M, O, L$.*

Theorem 3. (Unique Typing) *If $B \vdash s : M_1, O_1, L_1$ and $B \vdash s : M_2, O_2, L_2$, then $M_1 = M_2$ and $O_1 = O_2$ and $L_1 = L_2$.*

Theorem 4. (Subject Reduction) *For a program p, if $\vdash p : B$ and $B \vdash R : M, O, L$ and $R \to s'$, then there exists M', O', L' such that $B \vdash s' : M', O', L'$ and $M' \subseteq M$ and $O' \subseteq O$ and $L' \subseteq L$.*

Theorem 5. (Preservation) *For a program p, if $\vdash p : B$ and $B \vdash s : M, O, L$ and $s \longmapsto s'$, then there exists M', O', L' such that $B \vdash s' : M', O', L'$ and $M' \subseteq M$ and $O' \subseteq O$ and $L' \subseteq L$.*

Proof. From $s \longmapsto s'$ we have that there exist a context C and a redex R such that $s = C[R]$, and that there exists s'' such that $C[R] \longmapsto C[s'']$ and $R \to s''$. The proof proceeds by straightforward induction on C. □

For a statement s and a type environment B, we have from Theorem 2 and Theorem 3 that there exist unique M, O, L such that $B \vdash s : M, O, L$, so we define

$$\mathsf{MHP}^B_{type}(s) = M$$

We use the subscript *type* to emphasize that the definition is type based.

The following two theorems say that the type system gives a conservative approximation to the MHP computation problem, and an exact solution for programs without recursion.

Theorem 6. (Overapproximation) *For a program p, a statement s in p, and a type environment B such that $\vdash p : B$, we have $\mathsf{MHP}_{sem}(s) \subseteq \mathsf{MHP}^B_{type}(s)$.*

We patterned Theorem 6 after [11, Theorem 3]. In the case where s is the body of the main procedure, Theorem 6 says that $\mathsf{MHP}^B_{type}(s)$ is an overapproximation of the MHP information for the entire program.

The next theorem shows that there is no loss of precision in the type-based approach for programs without recursion. See Appendix B of the full version for a proof.

Theorem 7. (Equivalence) *For a program without recursion, where the body of main is the statement s, we have that there exists B such that $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}^B_{type}(s)$.*

Complexity. We can now state the complexity of the type-based approach.

Theorem 8. *For a program of size n, we can compute B and $\mathsf{MHP}^B_{type}(s)$ in $O(n \cdot \max(n, k))$ time, where $k = |\mathsf{MHP}^B_{type}(s)|$ is the size of the output produced by the type system.*

Proof. We first note that we can use the approach of Lee and Palsberg [11] to rephrase the problem of computing B and $\mathsf{MHP}^B_{type}(s)$ as the problem of finding the minimal solution to a collection of set constraints that are generated from the program text. For our type system, those set constraints are all of the forms:

$$l \in v \tag{20}$$

$$v \subseteq v' \tag{21}$$

$$symcross(v, v') \subseteq w \tag{22}$$

$$w \subseteq w' \tag{23}$$

Here v, v' range over sets of labels, while w, w' range over sets of pairs of labels. The maximal size of each set of labels is $O(n)$, the maximal size of each set

of pairs of labels is k (by definition), and the number of constraints is $O(n)$. We proceed by first solving the constraints of the forms (20) and (21) by a straightforward propagation-based algorithm akin to the one that Palsberg and Schwartzbach used to solve a related kind of set constraints [17]; this takes $O(n^2)$ time. Then we solve the constraints of the forms (22) and (23) by the same algorithm but this time we propagate pairs of labels rather than single labels; this takes $O(n \cdot k)$ time. In total, we spent $O(n \cdot \max(n, k))$ time. □

Since $k = O(n^2)$ in the worst case, we get a cubic algorithm, but our experiments show that k is $O(n)$ in practice.

When we combine Theorem 7 and Theorem 8, we get that we can solve the MHP computation problem for programs without recursion in $O(n \cdot \max(n, k))$ time, while we get a conservative approximation for programs with recursion.

Programs with Recursion. Theorems 6 and 7 indicate that some uses of recursion cause the type system to produce an approximate result rather than an accurate result. Specifically, our type system may be conservative if recursive calls introduce non-termination. For example, see the second program in Figure 1. The program has a loop with the statement $async\{a^1\}$ in the body so one might think that a^1 may happen in parallel with itself. However, the loop body also calls the procedure g that is non-terminating. So, the program execution will never get around to executing $async\{a^1\}$ a second time. In summary, for the second program in Figure 1, the MHP set is empty.

Let us now take a look at how the type system analyzes the second program in Figure 1. Let

$$B = [\ \ g \mapsto (\emptyset, \emptyset, \emptyset), \quad main \mapsto (\{(1,1)\}, \{1\}, \{1\})\ ]$$

From Rule (17) we have that to show that the entire program has type B, we must derive the following two judgments:

$$B \vdash \mathsf{body}(g) : \emptyset, \emptyset, \emptyset \tag{24}$$

$$B \vdash \mathsf{body}(main) : \{(1,1)\}, \{1\}, \{1\} \tag{25}$$

Let us consider those judgments in turn.

We have that $\mathsf{body}(g) = g()$ so Rule (16) gives us the judgment (24).

We have that $\mathsf{body}(main) = loop\ \{\ async\ \{\ a^1\ \};\ g()\ \}$ so from Rules (11), (12), (14), (16) we can produce this derivation that concludes with judgment (25):

$$\dfrac{\dfrac{\dfrac{B \vdash a^1 : \emptyset, \emptyset, \{1\}}{B \vdash async\ \{\ a^1\ \} : \emptyset, \{1\}, \{1\}} \qquad B \vdash g() : \emptyset, \emptyset, \emptyset}{B \vdash async\ \{\ a^1\ \};\ g() : \emptyset, \{1\}, \{1\}}}{B \vdash \mathsf{body}(main) : \{(1,1)\}, \{1\}, \{1\}}$$

In conclusion, the type system over-approximates non-termination and therefore concludes that a^1 may happen in parallel with itself.

5 An Algorithm for the MHP Decision Problem

We now give a linear-time algorithm for the MHP decision problem, even in the presence of recursion and potential non-termination. Our algorithm is based on constrained dynamic pushdown networks (CDPNs) [4], an infinite model of computation with nice decidability properties. Informally, CDPNs model collections of sequential pushdown processes running in parallel, where each process can "spawn" a new process or, under some conditions, observe the state of its children. We follow the presentation in [4].

Preliminaries. Let Σ be an alphabet, and let $\rho \subseteq \Sigma \times \Sigma$ be a binary relation on Σ. A set $S \subseteq \Sigma$ is *ρ-stable* if and only if for each $s \in S$ and for each $t \in \Sigma$, if $(s,t) \in \rho$ then t is also in S. A ρ-stable regular expression over Σ is defined inductively by the grammar:

$$e ::= S \mid e \cdot e \mid e^*$$

where S is a ρ-stable set. We derive a ρ-stable regular language from a ρ-stable regular expression in the obvious way and identify the expression with the language it denotes.

CDPNs. A *constrained dynamic pushdown network* (CDPN) [4] (A, P, Γ, Δ) consists of a finite set A of *actions*, a finite set P of *control locations*, a finite alphabet Γ of *stack symbols* (disjoint from P), and a finite set Δ of *transitions* of the following forms:

$$\phi : p\gamma \xrightarrow{a} p_1 w_1 \qquad \text{or} \qquad \phi : p\gamma \xrightarrow{a} p_1 w_1 \rhd p_2 w_2,$$

where $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, $a \in A$, $w_1, w_2 \in \Gamma^*$, and ϕ is a ρ_Δ-stable regular expression over P with

$$\rho_\Delta = \{\ (p,p') \in P \times P \mid \exists \psi : p\gamma \xrightarrow{a} p'w \text{ in } \Delta, \text{ or } \exists \psi : p\gamma \xrightarrow{a} p'w \rhd p''w' \text{ in } \Delta\ \}$$

The ρ-stable property guarantees that whenever a control location p is matched by an expression ϕ, all its successors' control locations are also matched.

Semantics. CDPN *configurations* model the execution states of CDPN instances. Intuitively, a configuration of a CDPN is a tree with each node marked with the configuration of a pushdown process, and the children of a node are configurations of pushdown processes spawned by it, which are ordered by age (the more recently spawned child is to the right). The configuration of each pushdown process models a single thread execution state in a parallel program, which includes control location describing the thread state and stack symbols modeling the stack storage. Formally, given a set $X = \{x_1, \ldots, x_n\}$ of variables, define the set $\mathcal{T}[X]$ of M-terms over $X \cup P \cup \Gamma$ as the smallest set satisfying:

(a) $X \subseteq \mathcal{T}[X]$;
(b) If $t \in \mathcal{T}[X]$ and $\gamma \in \Gamma$, then $\gamma(t) \in \mathcal{T}[X]$;

(c) For each $n \geq 0$, if $t_1, \ldots, t_n \in \mathcal{T}[X]$ and $p \in P$, then $p(t_1, \ldots, t_n) \in \mathcal{T}[X]$.

Notice that n can be zero in case (c); we often write p for the term $p()$. A *ground M-term* is an M-term without free variables. The set of ground M-terms is denoted $\mathcal{T}$.

We now define the semantics of CDPNs as a transition system. An M-configuration is a ground M-term; we write Conf^M to denote the set of M-configurations. We define a context C as a M-term with one free variable, which moreover appears at most once in the term. If t is a ground M-term, then $C[t]$ is the ground M-term obtained by substituting the free variable with t.

The M-configuration $\gamma_m \ldots \gamma_1 p(t_1, \ldots, t_n)$, for $n, m \geq 0$ represents a process in control location p and $\gamma_m \ldots \gamma_1$ on the stack (with γ_1 on top), which has spawned n child processes. The ith child, along with all its descendants, is given by t_i. The child processes are ordered so that the rightmost child t_n is latest spawned. We call $\gamma_m \ldots \gamma_1 p$ the topmost process in the M-configuration.

The semantics of a CDPN is given as a binary transition relation $\rightarrow_M$ between M-configurations. Given an M-configuration t of one of the forms $\gamma_m \ldots \gamma_1 p(t_1, \ldots, t_n), n \geq 1$ or $\gamma_m \ldots \gamma_1 p$, we define $root(t)$ to be the control location p of the topmost process in t. We define $\rightarrow_M$ as the smallest relation such that the following hold:

(a) if $(\phi : p\gamma \xrightarrow{a} p_1 w_1) \in \Delta$ and $root(t_1) \ldots root(t_n) \in \phi$, then $C[\gamma p(t_1, \ldots, t_n)] \rightarrow_M C[w_1^R p_1(t_1, \ldots, t_n)]$; and
(b) if $(\phi : p\gamma \xrightarrow{a} p_1 w_1 \rhd p_2 w_2) \in \Delta$ and $root(t_1) \ldots root(t_n) \in \phi$, then $C[\gamma p(t_1, \ldots, t_n)] \rightarrow_M C[w_1^R p_1(t_1, \ldots, t_n, w_2^R p_2)]$.

Intuitively, transitions between M-configurations model parallel program execution. With a CDPN transition rule $\phi : p\gamma \xrightarrow{a} p_1\omega_1$, a process in the M-configuration steps to its next state and updates its stack; with a CDPN transition rule $\phi : p\gamma \xrightarrow{a} p_1\omega_1 \rhd p_2\omega_2$, a process in the M-configuration spawns a new pushdown process as its newest child. The constraint ϕ in a transition rule provides a simple way to communicate between the parent process and its children. For example, given control location $\natural \in P$ standing for termination state, a parent process cannot step over a transition rule $\natural^* : p\gamma \xrightarrow{a} p_1\omega_1$ until all its children have terminated.

Given the transition relation $\rightarrow_M$, we define the operators pre and pre^* on sets of M-configurations in the standard way.

Regular Sets of M-configurations. We define M-tree automata that accept a set of M-configurations. Formally, an M-tree automaton (Q, F, δ) consists in a finite set Q of states, a set $F \subseteq Q$ of final states, and a set δ of rules of the following two forms: (a) $\gamma(q) \rightarrow q'$, where $\gamma \in \Gamma$, and $q, q' \in Q$, and (b) $p(L) \rightarrow q$ where $p \in P$, $q \in Q$, and L is a regular language over Q. We define the relation $\rightarrow_\delta$ between terms over $P \cup \Gamma \cup Q$ as: $t \rightarrow_\delta t'$ if and only if there exists a context C, statements s, s', and a rule $r \in \delta$ such that $t = C[s]$, $t' = C[s']$, and (a) either $r = \gamma(q) \rightarrow q'$ and $s = \gamma(q)$ and $s' = q'$, or (b) $r = p(L) \rightarrow q$, $s = p(q_1, \ldots, q_n)$, $q_1 \ldots q_n \in L$, and $s' = q$. A term t is accepted by the M-tree automaton A^M

denoted as $t \in L(A^M)$ if $t \to_\delta^* q$ for some $q \in F$, where $\to_\delta^*$ is the reflexive transitive closure of $\to_\delta$. The language of an M-tree automaton is the set of all M-terms accepted by it.

From X10 to CDPNs. We now give a translation from programs in our syntax to CDPNs. Our translation starts with a control-flow graph (CFG) representation of a program, in which each procedure f is represented as a labeled, directed graph $G_f = (V_f, E_f, \mathit{entry}_f, \mathit{exit}_f)$ where V_f is a set of control nodes, $E_f \subseteq V_f \times \mathsf{ops} \times V_f$ is a set of labeled directed edges labeled by operations from ops (defined below), and entry_f and exit_f are nodes in V_f denoting the entry and exit nodes of a CFG. Each edge label is either a labeled action a^l, a call $\mathrm{call}(g)$ to a procedure g, an asynchronous call $\mathrm{async}(g)$ to a procedure g, or a finish $\mathrm{finish}(g)$ to a procedure g. A control flow graph representation can be computed from the program syntax using standard compiler techniques [2].

Additionally, we make the simplifying assumption that each label l is used at most once, and that if the primitive statement a^l is translated to the edge (u, a^l, v), then the node u has no other outgoing edges. Thus, the node u uniquely determines the label l which is about to be executed, and we can identify node u with l.

As usual, we assume $V_f \cap V_g = \emptyset$ for two distinct procedures f, g. Let

$$V = \cup\{\ V_f \mid f \text{ is a procedure }\} \qquad E = \cup\{\ E_f \mid f \text{ is a procedure }\}.$$

We now define a CDPN $M_\mathcal{G}$ from a CFG representation $\mathcal{G}$. The set of actions consists of all actions a in ops, together with a new "silent" action τ. The set of control locations $P = \{\#, \natural\}$. The set of stack symbols $\Gamma = V \cup \{\mathrm{wait}[u, g, v] \mid (u, \mathrm{finish}(g), v) \in E\} \cup \{\$\}$. Intuitively, we will use the stack to maintain the program stack, with the topmost symbol being the current program point. The control location $\#$ is the dummy location used to orchestrate program steps, and the control location $\natural$ is used to indicate the process execution has terminated. We shall implicitly assume that each stack has a bottom symbol \$.

Now for the transitions in Δ. For each $(u, a, v) \in E$, we have the rule $P^* : \# u \xrightarrow{a} \# v$. For each edge $(u, \mathrm{call}(g), v)$, we have the rule $P^* : \# u \xrightarrow{\tau} \# \mathit{entry}_g\, v$. For each edge $(u, \mathrm{async}(g), v)$, we have the rules $P^* : \# u \xrightarrow{\tau} \# v \rhd \# \mathit{entry}_g$. To model returns from a procedure g, we add the rule $P^* : \# \mathit{exit}_g \xrightarrow{\tau} \#$. For each edge $(u, \mathrm{finish}(g), v)$, we add the rule $P^* : \# u \xrightarrow{\tau} \# \mathrm{wait}[u, g, v] \rhd \# \mathit{entry}_g$.

Next, we give the rules performing the synchronization at the end of a finish statement. The first rule, $\natural^* : \# \$ \xrightarrow{\tau} \natural$, encodes that a process on its last stack symbol "\$" goes to the control state $\natural$ when all its children terminated. The second rule, $P^* \natural : \# \mathrm{wait}[u, p, v] \xrightarrow{\tau} \# v$, encodes that the "top level" finish call finishes when the entire finish (spawned as its youngest child) finishes. These two rules ensure that a process makes progress beyond a $\mathrm{finish}(g)$ statement only when all processes spawned transitively from g terminate.

It is easy to see that for every CFG $\mathcal{G}$, the CDPN $M_\mathcal{G}$ preserves all the behaviors of $\mathcal{G}$. Moreover, $M_\mathcal{G}$ is *linear* in the size of $\mathcal{G}$.

Solving the MHP decision problem. We solve the MHP decision problem by performing a reachability test between the initial program M-configuration and a family of interesting M-configurations. In particular, for the MHP problem given two labels l and l', we are interested in the family of M-configurations $\mathsf{Conf}^M_{l,l'}$ in which there exists two processes, one about to execute l and the other about to execute l'. Formally, for edges $l \xrightarrow{a} v$, $l' \xrightarrow{a'} v'$ on $\mathcal{G}$ with labels l, l' and primitive statements a, a', we define M-configuration $c \in \mathsf{Conf}^M_{l,l'}$ if and only if there exists two processes in c of the form $\gamma l(p(t_1, \ldots, t_n))$ and $\gamma' l'(p(t'_1, \ldots, t'_m))$ in c where $\gamma, \gamma' \in \Gamma^*$, $t_1, \ldots, t_n$, $t'_1, \ldots, t'_m$ are ground M-terms. Both processes have program points l, l' on the top of the stacks, and thus, l and l' may happen in parallel.

We now give a M-tree automaton $A^M_{l,l'}$ that can recognize exactly the M-configurations in $\mathsf{Conf}^M_{l,l'}$. Given CDPN M with two labels l, l' (program points on $\mathcal{G}$), we define the M-tree automaton $A^M_{l,l'} = (Q, F, \delta)$ as follow. The state set is defined as

$$Q = Q_p \cup Q_r$$

where two symmetric subsets

$$Q_p = \{q_{\mathsf{p}00}, q_{\mathsf{p}10}, q_{\mathsf{p}01}, q_{\mathsf{p}11}\} \qquad Q_r = \{q_{\mathsf{r}00}, q_{\mathsf{r}10}, q_{\mathsf{r}01}, q_{\mathsf{r}11}\}$$

give all states for P-transitions and Γ-transitions. We define q_{pi} as the i-th state in Q_p, and q_{ri} as the i-th state in Q_r for $i = 1, 2, 3, 4$. The 4 states in both sets Q_p and Q_r with tags $\mathsf{00}, \mathsf{10}, \mathsf{01}, \mathsf{11}$ on subscripts give the intuitive meanings that neither stack symbol l nor l' has been recognized yet, stack symbol l has been recognized (the first bit is set), stack symbol l' has been recognized (the second bit is set), both stack symbol l and stack symbol l' have been recognized (both bits are set). The terminal state set is defined as

$$F = \{q_{\mathsf{r}11}, q_{\mathsf{p}11}\}.$$

The transition rule set is defined as

$$\begin{aligned}\delta = \{&p() \to q_{\mathsf{p}00}\ ,\ l(q_{\mathsf{p}01}) \to q_{\mathsf{r}11}\ ,\ l'(q_{\mathsf{p}10}) \to q_{\mathsf{r}11}\ ,\ l(q_{\mathsf{p}00}) \to q_{\mathsf{r}10}\ ,\\ &l'(q_{\mathsf{p}00}) \to q_{\mathsf{r}01}\ ,\ p(Q^*, q_{\mathsf{10}}, Q^*, q_{\mathsf{01}}, Q^*) \to q_{\mathsf{p}11}\ ,\\ &p(Q^*, q_{\mathsf{01}}, Q^*, q_{\mathsf{10}}, Q^*) \to q_{\mathsf{p}11}\ ,\ \gamma(q_i) \to q_{ri}\ ,\ p(Q^*, q_i, Q^*) \to q_{pi}\}.\end{aligned}$$

In the transition rule set above, notice that q_i is the state in $\{q_{ri}, q_{pi}\}$, and similarly $q_{\mathsf{00}}, q_{\mathsf{10}}, q_{\mathsf{01}}, q_{\mathsf{11}}$ are states in $\{q_{\mathsf{r}00}, q_{\mathsf{p}00}\}$, $\{q_{\mathsf{r}10}, q_{\mathsf{p}10}\}$, $\{q_{\mathsf{r}01}, q_{\mathsf{p}01}\}$, $\{q_{\mathsf{r}11}, q_{\mathsf{p}11}\}$ respectively; $\gamma \in \Gamma$ is an arbitrary stack symbol; and $p \in P$ is an arbitrary control location. We will follow this convention in the rest of this paper.

It is easy to perform a bottom up scan on any M-configuration t with M-tree automaton $A^M_{l,l'}$. The M-tree automaton $A^M_{l,l'}$ recognizes t if and only if there are two processes in t running at program points l, l' in parallel. To be noted that the M-configuration t is not necessary a valid program configuration to be recognized by $A^M_{l,l'}$ as long as it belongs to $\mathsf{Conf}^M_{l,l'}$. A valid program configuration means the configuration is reachable from the initial program configuration by execution. The following theorem is proved in Appendix C of the full version.

Theorem 9. $\mathsf{Conf}^M_{l,l'} = L(A^M_{l,l'})$.

Algorithm and complexity. The key to our decision procedure is the following main result of [4].

Theorem 10. *[4] For every CDPN M, and for every M-tree automaton A, there is an effective procedure to construct an M-tree automaton A^* such that $L(A^*) = \mathsf{pre}^*(L(A))$.*

The procedure in [4] applies backward saturation rules to the automaton A. Given a CFG $\mathcal{G}$, the MHP decision problem is solved by:

(a) Constructing CDPN $M_{\mathcal{G}}$ and M-tree automaton $\mathcal{A}^M_{l,l'}$;
(b) Finding the pre^*-image of $L(\mathcal{A}^M_{l,l'})$ using Theorem 10, and checking if the initial configuration $\#entry_{main}$ is in $\mathsf{pre}^*(L(A^M_{l,l'}))$.

Step (a) can be performed in time linear in the size of the input $\mathcal{G}$. The M-tree automaton $A^M_{l,l'}$ is clearly constant and independent of the input program. A careful observation of the construction in [4] shows that (b) is also linear. Thus, we have the following theorem.

Theorem 11. *The MHP decision problem can be solved in linear time in the size of a program.*

Appendix A of the full version gives a detailed example of the CDPN-based approach applied to the first program in Figure 1.

6 Solving the MHP Computation Problem

We can compute all pairs of statements that may happen in parallel with this two-step algorithm:

1. Run the type-based analysis (Section 4) and produce a set of candidate pairs.
2. For each of the candidate pairs, run the CDPN-based decision procedure (Section 5), and remove those pairs that cannot happen in parallel.

Theorem 12. *For a program of size n, for which the type-based analysis produces k candidate pairs, the MHP computation problem can be solved in $O(n \cdot \max(n,k))$ time.*

Proof. Theorem 8 says that Step 1 runs in $O(n \cdot \max(n,k))$ time, and Theorem 11 implies that Step 2 runs in $O(n \cdot k)$ time because we apply an $O(n)$ algorithm k times. The total run time of the two-step algorithm is $O(n \cdot \max(n,k)) + O(n \cdot k)$ $= O(n \cdot \max(n,k))$. □

	Static counts		Data-race detection			Analysis time (ms)		
Benchmarks	LOC	#async	MHP	MHP +Types	MHP+Types +Andersen	Step:1	Steps:1+2	All-pairs
stream	70	4	160	21	9	7	33	318
sor	185	7	31	8	3	16	21	169
series	290	3	3	3	3	11	13	237
sparsemm	366	4	55	4	2	14	52	2,201
crypt	562	2	366	100	100	54	164	2,289
moldyn	699	14	31882	880	588	43	14,992	57,308
linpack	781	8	67	33	33	14	60	5,618
mg	1,858	57	4884	431	421	69	9,970	114,239
mapreduce	53	3	3	2	2	3	16	78
plasma	4,623	151	8475	2084	760	503	36,491	5,001,310

Fig. 4. Data-race detection.

7 Experimental Results

We now show experimental results that show (1) how to use our MHP analysis for race detection and (2) how much time is spent on the two parts of the algorithm in Section 6. We ran our experiments on a Apple iMac with Mac OS X and a 2.16 GHz Intel Core 2 Duo processor and 1 Gigabyte of memory.

Benchmarks. We use 10 benchmarks taken from the HPC challenge benchmarks (stream), the Java Grande benchmarks in X10 (sor, series, sparsemm, crypt, moldyn, linpack), the NAS benchmarks (mg), and two benchmarks written by ourselves (mapreduce, plasma). In Figure 4, columns 2+3 show the number of lines of code (LOC) and the number of asyncs. The number of asyncs includes the number of foreach and ateach loops, which are X10 constructs that let all the loop iterations run in parallel. We can think of foreach and ateach as plain loops where the body is wrapped in an async. Our own plasma simulation benchmark, called plasma, is the longest and by far the most complicated benchmark with 151 asyncs and 84 finishes. None of the benchmarks use recursion! In particular, none of the benchmarks use the problematic programming style illustrated in the second program in Figure 1.

Measurements. In Figure 4, columns 4–6 show results from doing race detection on our 10 benchmarks. The column MHP shows the number of pairs of primitive statements that read or write nonlocal variables that our analysis found may happen in parallel. Given that none of the benchmarks use recursion, we needed to use only Step 1 of the algorithm in Section 6.

The MHP analysis problem is all about control flow. We complement the control-flow analysis with two data flow analyses, one that uses types and one that uses pointer analysis. The column Type Check refines the MHP column by allowing only pairs of statements for which the accesses are to variables of the same type. The column Andersen Algo refines the Type Check column by

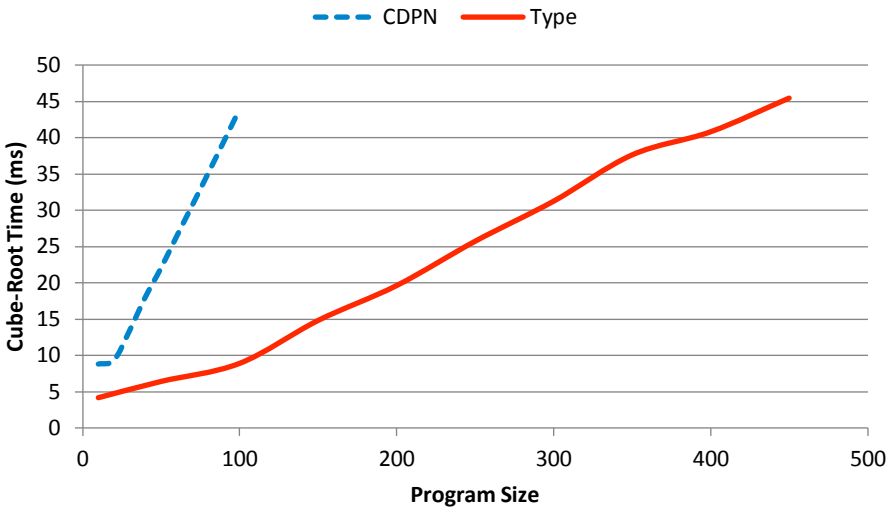


Fig. 5. The cubic root of the analysis time for the third program in Figure 1.

allowing only pairs of statements for which Andersen's pointer analysis algorithm finds that the statements may access the same variable.

Note that we can give an alternative and slower algorithm for the MHP computation problem by running the CDPN-based decision procedure on all possible pairs. In Figure 4, column 7 shows the analysis time for the type-based Step 1 that is sufficient for our benchmarks, column 8 shows how long it would take to run Step 1+2 in case we were unable to determine that Step 2 was unnecessary, and column 9 shows the analysis times for the CDPN-based decision procedure on all pairs. For all benchmarks, it much faster to run Step 1 only rather than Step 1+2, which, in turn, is much faster than to run the decision procedure on all pairs.

Assessment. The combination of the control-flow-oriented MHP analysis and the data-flow-oriented Type Check and Andersen's algorithm is powerful. The final column in Figure 4 contains numbers that are low enough that they are a good starting point for other analyses or testing techniques that depend on MHP information. One such approach is the Race Directed Random Testing of Sen [20] that needs MHP information as a starting point.

Scalability. Our X10 benchmarks form one extreme for the algorithm in Section 6: Step 2 isn't needed at all for those benchmarks. Let us now consider the other extreme where Step 1 provides no savings because the program is recursive and the size of the output is $O(n^2)$. Our question is then: how much time is spent on Step 1 and how much time is spent on Step 2? As our benchmarks, we will use the family of programs that are shown as the third program in Figure 1. For each N, we have one such program. The Nth program contains N procedures

that call each other recursively in a closed chain. The main procedure executes a parallel loop that calls the 0'th procedure.

Our experiments show that the running times for the type-based Step 1 grow more slowly than the running times for the CDPN-based Step 2. The type-based Step 1 can handle $N = 450$ within 100 seconds, while for $N = 100$, the CDPN-based Step 2 takes a lot more than 100 seconds. Figure 5 shows the cubic-root of the analysis time for N up to 500. The near-linear curves in Figure 5 suggest that both steps use cubic time for the third program in Figure 1. Two linear regressions on the data in Figure 5 lead to these formulas for the running times:

$$\begin{array}{ll} \text{Type-based Step 1:} & time(n) = .00109 \times n^3 + \ldots \\ \text{CDPN-based Step 2:} & time(n) = .06943 \times n^3 + \ldots \end{array}$$

The constant in front of n^3 is more than 63 times bigger for Step 2 than for Step 1 so in the worst case Step 2 dwarfs Step 1.

8 Conclusion

We have presented two algorithms for static may-happen-in-parallel analysis of X10 programs, including a linear-time algorithm for the MHP decision problem and a two-step algorithm for the MHP computation problem that runs in $O(n \cdot \max(n, k))$ time, where k is a statically determined upper bound on the number of pairs that may happen in parallel. Our results show that the may-happen-in-parallel analysis problem for languages with async-finish parallelism is computationally tractable, as opposed to the situation for concurrent languages with rendezvous or locks. Our results are applicable to various forms of parallelism and synchronization, including fork-join parallelism.

Acknowledgements. This material is based upon research performed in collaborative facilities renovated with funds from the National Science Foundation under Grant No. 0963183, an award funded under the American Recovery and Reinvestment Act of 2009 (ARRA).

References

1. Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, pages 183–193. ACM, 2007.
2. Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.
3. Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC*, pages 152–169, 2005.
4. Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, pages 473–487, 2005.
5. Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented

approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.
6. Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.
7. Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Symposium on Testing, Analysis, and Verification*, pages 36–48, 1991.
8. Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *Proceedings of VMCAI'11, Verification, Model Checking, and Abstract Interpretation*, pages 199–213, 2011.
9. Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *LICS'09, 24th Annual Symposium on Logic in Computer Science*, pages 27–36, 2009.
10. Peter Lammich and Markus Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation and procedures. In *Proceedings of CONCUR'07*, pages 287–302, 2007.
11. Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPOPP'10, 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.
12. Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent Java programs. In *LCPC*, pages 194–208, 2004.
13. Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *PPOPP*, pages 129–138, 1993.
14. Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of POPL'07, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2007.
15. Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *SIGSOFT FSE*, pages 24–34, 1998.
16. Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing HP information for concurrent Java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.
17. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
18. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.
19. Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. In *Proceedings of ESOP'00, European Symposium on Programming*, pages 351–365. Springer-Verlag (*LNCS* 1782), 2000.
20. Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of PLDI'08, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, Tucson, Arizona, June 2008.
21. Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

Appendix A: CDPN Example

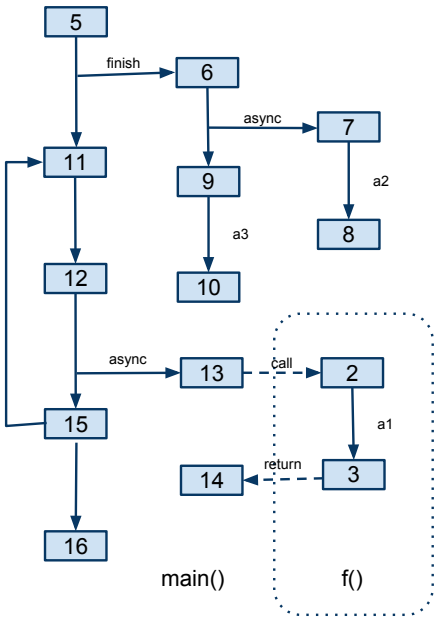


Fig. 6. Control flow graph for the first program in Figure 1.

In this example, we will walk though the CDPN method to solve the MHP decision problem for the first program in Figure 1.

Figure 6 gives the control flow graph $\mathcal{G}$ for first program in Figure 1. Each node in the graph represents one unique program point between two statements. We use the line number of the direct following statement in the sample code as the unique ID of that program point, which is labeled on the node. Edges on the graph represent statements (or actions) in the program. There are some special 3-way edges, representing statements async, and finish, since these statements will have side effects of spawning new threads. One ending point of the 3-way edge will connect to the next program point in the parent control flow, and the other one will point to the starting point of the new spawned control flow. We use dash lines to describe control flows of procedure call, and the calling path and returning path are explicitly put on the graph.

Following the instructions in Section 5, it is straightforward to translate the control flow graph $\mathcal{G}$ in Figure 6, into CDPN $M_{\mathcal{G}}$. Program point IDs are used as CDPN stack symbols, which provide execution location information for push-down processes in CDPN. Figure 7 gives all the $M_{\mathcal{G}}$ transition rules translated from $\mathcal{G}$. Noticed that, an extra stack symbol $wait[5, 6, 11]$ is inserted in tran-

1. $P* : \#\,2 \xrightarrow{a_1} \#\,3$
2. $P* : \#\,3 \xrightarrow{\tau} \#$
3. $P* : \#\,5 \xrightarrow{\tau} \#\,wait[5,6,11] \rhd \#\,6$
4. $P* : \#\,6 \xrightarrow{\tau} \#\,9 \rhd \#\,7$
5. $P* : \#\,7 \xrightarrow{a_2} \#\,8$
6. $\natural* : \#\,8 \xrightarrow{\tau} \natural$
7. $P* : \#\,9 \xrightarrow{a_3} \#\,10$
8. $\natural* : \#\,10 \xrightarrow{\tau} \natural$
9. $P* : \#\,11 \xrightarrow{\tau} \#\,12$
10. $P* : \#\,12 \xrightarrow{\tau} \#\,15 \rhd \#\,13$
11. $P* : \#\,13 \xrightarrow{\tau} \#\,2\,14$
12. $\natural* : \#\,14 \xrightarrow{\tau} \natural$
13. $P* : \#\,15 \xrightarrow{\tau} \#\,16$
14. $P * \natural : \#\,wait[5,6,11] \xrightarrow{\tau} \#\,11$
15. $\natural* : \#\,16 \xrightarrow{\tau} \natural$

Fig. 7. CDPN transitions for the first program in Figure 1.

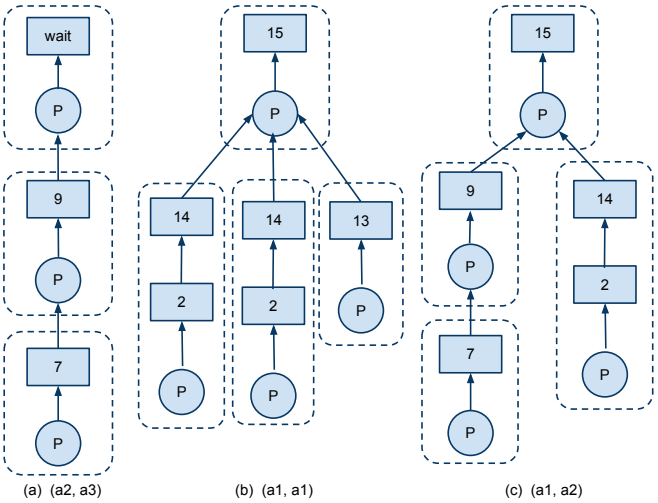


Fig. 8. M-configurations for the first program in Figure 1.

Row	CDPN Rule in Fig. 7	Pre-conditions	M-Tree Automaton Transitions
1	$\emptyset$	$\emptyset$	$2(q^p_{\mathsf{p}00}) \to q^p_{\mathsf{r}10}$ (26) $7(q^p_{\mathsf{p}00}) \to q^p_{\mathsf{r}01}$ (27) $2(q^p_{\mathsf{p}01}) \to q^p_{\mathsf{r}11}$ (28) $7(q^p_{\mathsf{p}10}) \to q^p_{\mathsf{r}11}$ (29) $p() \to q^p_{\mathsf{p}00}$ (30) $p'(Q*, q^P_{10}, Q*, q^P_{01}, Q*) \to q^{p'}_{\mathsf{p}11}$ (31) $p'(Q*, q^P_{01}, Q*, q^P_{10}, Q*) \to q^{p'}_{\mathsf{p}11}$ (32) $\gamma(q^p_i) \to q^p_{ri}$ (33) $p'(Q*, q^p_i, Q*) \to q^{p'}_{pi}$ (34)
2	Rule 6	(30): $\natural() \to q^{\natural}_{\mathsf{p}00}$, (33): $(q^{\natural}_{\mathsf{p}00}) \to q^{\natural}_{\mathsf{r}00}$	$8(\#()) \to^* q^{\natural}_{\mathsf{r}00}$ (35)
3	Rule 5	(35): $8(\#()) \to^* q^{\natural}_{\mathsf{r}00}$	$7(\#()) \to^* q^{\natural}_{\mathsf{r}00}$ (36)
4	Rule 8	(30): $\natural() \to q^{\natural}_{\mathsf{p}00}$, (33): $(q^{\natural}_{\mathsf{p}00}) \to q^{\natural}_{\mathsf{r}00}$	$10(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$ (37)
5	Rule 7	(37): $10(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$	$9(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$ (38)
6	Rule 4	(30): $\#() \to q^{\#}_{\mathsf{p}00}$ (27): $7(q^{\#}_{\mathsf{p}00}) \to q^{p}_{\mathsf{r}01}$ (34): $\#(q^{P}_{\mathsf{r}01}) \to q^{\#}_{\mathsf{p}01}$ (33): $9(q^{\#}_{\mathsf{p}01}) \to q^{\#}_{\mathsf{r}01}$	$6(\#(Q^{p}*)) \to^* q^{\#}_{\mathsf{r}01}$ (39)
7	Rule 4	(36): $7(\#()) \to^* q^{\natural}_{\mathsf{r}00}$ (38): $9(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$	$6(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$ (40)
8	Rule 11	(30): $\#() \to q^{\#}_{\mathsf{p}00}$ (26): $2(q^{\#}_{\mathsf{p}00}) \to q^{\#}_{\mathsf{r}10}$ (33): $14(q^{\#}_{\mathsf{p}10}) \to q^{\#}_{\mathsf{r}10}$	$13(\#()) \to^* q^{\#}_{\mathsf{r}10}$ (41)
9	Rule 10	(41): $13(\#()) \to^* q^{\#}_{\mathsf{r}10}$ (34): $\#(q^{P}_{\mathsf{r}10}) \to q^{\#}_{\mathsf{p}10}$ (33): $15(q^{\#}_{\mathsf{p}10}) \to q^{\#}_{\mathsf{r}10}$	$12(\#(Q^{p}*)) \to^* q^{\#}_{\mathsf{r}10}$ (42)
10	Rules 9, 14	(42): $12(\#(Q^{P}*)) \to^* q^{\#}_{\mathsf{r}10}$	$wait[5,6,11](\#(Q^P * q^{\natural})) \to^* q^{\#}_{\mathsf{r}10}$ (43)
11	Rule 3	(40): $6(\#(Q^{\natural}*)) \to^* q^{\natural}_{\mathsf{r}00}$ (43): $wait[5,6,11](\#(Q^P * q^{\natural})) \to^* q^{\#}_{\mathsf{r}10}$	$5(\#(Q^{\natural}*)) \to^* q^{\#}_{\mathsf{r}10}$ (44)
12	Rule 3	(39): $6(\#(Q^{p}*)) \to^* q^{\#}_{\mathsf{r}01}$ (34): $\#(q^{P}_{\mathsf{r}01}) \to q^{\#}_{\mathsf{p}01}$ (33): $wait[5,6,11](q^{\#}_{\mathsf{p}01}) \to q^{\#}_{\mathsf{r}01}$	$5(\#(Q^{P}*)) \to^* q^{\#}_{\mathsf{r}01}$ (45)

Fig. 9. Automaton for MHP(a^1, a^2).

sition rules 3 and 13 as synchronization point for finish statement. There are non-trivial constraints Φ in front of transition rules 6, 8, 12, 14, 15. Particularly, constraints $\natural*$ for transition rules 6, 8, 12, 15 are used to suspend process transitions from last statement to terminated state until all children have already fully terminated. Constraint $P*\natural$ for rule 14 will pause the finish statement until its most recent spawned child terminated. Each process in this model will not enter terminated state until all its children do, though it may have semantically terminated.

Figure 8 gives 3 examples of visualized M-configurations for CDPN $M_{\mathcal{G}}$ given above. M-configurations on the graph are trees with box and circle nodes. Box node represents control location, and circle node represents stack symbol in M-configuration. All the arrows on the control location nodes point to their top most stack symbol node, and the arrows on the stack symbol nodes point to lower stack symbol nodes or control location nodes in their parent processes. We use dash-line boxes to isolate configurations different processes in M-configurations. Sub-graphs (a), (b), (c) respectively depicted one of the M-configurations for actions (a^2, a^3), (a^1, a^1), (a^1, a^2) running in parallel. For example in the Sub-graph (a), the given M-configuration describes configurations of three processes. These configurations, from top to bottom, are $wait[5, 6, 11](p())$, $9(p())$ and $7(p())$, and they indicate that top most process is running at synchronization point wait[5,6,11], the process spawned by top most process is about to execute action a^3 (at point 9), and the leaf process spawned by $9(p())$ is about to execute a^2 (at point 7).

To be noted that not all of the M-configurations can be reached from the initial M-configuration. Here we select action points (a^1, a^2) for demonstrating how to solve MHP decision problem by backward reachability test on M-configurations.

We will construct an M-tree automaton $A^M_{a^1,a^2}$ for recognizing the family of M-configurations $\mathsf{Conf}^M_{a^1,a^2}$ with (a^1, a^2) two points running in parallel for the first program in Figure 1. Following that, we expend it into the M-tree automaton $A'^M_{a^1,a^2} = pre^*(A^M_{a^1,a^2})$ which recognizes the pre^*-images of $\mathsf{Conf}^M_{a^1,a^2}$. We will show that pair (a^1, a^2) do not MHP since the M-tree automaton $A'^M_{a^1,a^2}$ cannot recognize the initial M-configuration conf^M_{init} and the family of M-configurations with (a^1, a^2) running in parallel will never be reached.

In Figure 6, a^1, a^2 are on the program points 2 and 7 respectively. We instantiate M-tree automaton transition rules δ in $A^M_{a,a'}$ by substituting stack symbol a, a' with 2 and 7 for recognizing (a^1, a^2). M-configuration (c) in Figure 8 is one example of the M-configurations with (a^1, a^2) running in parallel, which can be

recognized by following transition rules:

$$\#() \to q_{\mathsf{p00}} \tag{46}$$
$$2(q_{\mathsf{p00}}) \to q_{\mathsf{r10}} \tag{47}$$
$$7(q_{\mathsf{p00}}) \to q_{\mathsf{r01}} \tag{48}$$
$$14(q_{\mathsf{r10}}) \to q_{\mathsf{r10}} \tag{49}$$
$$\#(q_{\mathsf{01}}) \to q_{\mathsf{p01}} \tag{50}$$
$$9(q_{\mathsf{01}}) \to q_{\mathsf{r01}} \tag{51}$$
$$\#(q_{\mathsf{r01}}, q_{\mathsf{r10}}) \to q_{\mathsf{p11}} \tag{52}$$

where rule (47), (48) discovering symbols a and a' are on the stack and (52) generates the terminal state by reading its two children threads with a^1 and a^2 running in parallel.

Before applying backward saturation rules, we propagate all control locations in automaton transition rules into their states so that automaton states will reflect control locations of the M-terms they recognized. Particularly, the automaton state with control $q \in Q$ location $p \in P$ is denoted as q^p and we transform automaton transition rule set from Γ to Γ^P by following 2 rules.

(a) If $p(Q*) \to q$ in Γ then $p(Q) \to q^p$ in Γ^P.
(b) If $\gamma(Q^p) \to q$ in Γ then $\gamma(Q^p) \to q^p$ in Γ^P.

As we mentioned before, there is a special control location $\natural$ in CDPN denoting process termination, and all M-configurations with their children processes terminated will be recognized by the automaton state $q^\natural$. Transition rule $p(Q^\natural *) \to q^p$ intuitively will recognize M-terms with all their children processes terminated.

Figure 9 shows the automaton for MHP(a^1, a^2). Let us explain the details of the construction. We denote state q with propagated control location $p \in P$ as q^p. Action a^1 has stack symbol $a = 2$, and action a^2 has stack symbol $a' = 7$. All control locations have been propagated into automaton states.

Row 1 in Figure 9 shows the initial M-tree automaton transition rules for recognizing (a^1, a^2) running in parallel. Rule (26) to rule (29) recognize stack symbols 2 and 7 for actions a^1 and a^2; rule (30) generate a trivial state q^p_{p00} for all leaf nodes in the M-configuration; rule (31) and rule (32) recognize configurations with children processes having a^1 and a^2 running in parallel; rule (33) and rule (34) propagate recognized states from bottom to top.

Row 2 in Figure 9 shows the inverse of the *terminate* operation at program point 8. Row 3 shows the inverse of the a^2 action for the control flow with $\natural$ control location. Row 4 shows the inverse of the *terminate* operation at program point 10. The new generated transition rule requires that all its children are in the $\natural$ control location. Row 5 shows the inverse of the a^3 action for the control flow with $\natural$ control location. Row 6 shows the inverse of the *async* statement at program point 6 for the control flow with # control location. Row 7 shows the inverse of the *async* statement at program point 6 for the control flow with $\natural$

control location. Row 8 shows the inverse of the *call f()* statement which includes action a^1. Row 9 shows the inverse of the *async* statement at program point 12 for the control flow with # control location. Row 10 shows the inverse of the *wait* operation at program point $wait[5, 6, 11]$, which requires that all its newest child has $\natural$ control location. Row 11 shows the inverse of the *finish* statement at program point 5 for the control flow; requires $\natural$ control location. Row 12 shows the inverse of the *finish* statement at program point 5 for the control flow NOT require $\natural$ control location. This concludes our explanation of Figure 9.

The initial M-configuration conf^M_{init} for the first program in Figure 1 is $5(\#)$ as listed in the beginning of the $\mathcal{G}$ in Figure 6. We will determine whether M-configurations with (a^1, a^2) running in parallel are reachable from the M-configuration $5(\#)$ by using pre^*-image of the M-tree automaton constructed above to recognize $5(\#)$.

We construct the pre^*-image of the M-tree automaton by applying backward saturation rules given in [4] on CDPN transition rules in (7) and M-tree automaton transition rules given in Row 1 in Figure 9. This process intuitively is executing the program backward and find all reachable previous M-configurations. The pre^*-image of the M-tree automaton will be built iteratively, and in each step, one CDPN transition rule will be selected, and constant number of new automaton transition rules will be inserted based on some existing automaton transition rules.

To simplify the notation we use $\gamma(p(q_1^{p'}, \ldots, q_n^{p'})) \rightarrow^* q^p$ for automaton transition rules $\gamma(q_p^L) \rightarrow q^p$ and $p(q_1^{p'}, \ldots, q_n^{p'}) \rightarrow q_p^L$ where q_p^L is the new created state in the follow illustration.

For CDPN transition rule 6: $\natural * : \#\, 8 \rightarrow \natural$ in Figure 7, we will add M-automaton transition rules

$$8(\#()) \rightarrow^* q^{\natural}_{\mathsf{r00}} \tag{53}$$

$$8(\#(Q^{\natural}*, q_i^{\natural}, Q^{\natural})) \rightarrow^* q^{\natural}_{pi} \tag{54}$$

$$8(\#(Q^{\natural}*, q^{\natural}_{\mathsf{10}}, Q^{\natural}*, q^{\natural}_{\mathsf{01}}, Q^{\natural}*)) \rightarrow^* q^{\natural}_{\mathsf{p11}} \tag{55}$$

$$8(\#(Q^{\natural}*, q^{\natural}_{\mathsf{01}}, Q^{\natural}*, q^{\natural}_{\mathsf{10}}, Q^{\natural}*)) \rightarrow^* q^{\natural}_{\mathsf{p11}} \tag{56}$$

based on the backward saturation rule R1 and M-tree automaton transitions (30),(34), (31), (32), (33) in Row 1 of Figure 9.

New M-tree automaton transition rules (53) to (56) will enable automaton state $q^{\natural}$ with *terminate* control location recognize running processes with stack symbol 8 due to the inverse execution of CDPN rule $\natural * : \#\, 8 \rightarrow \natural$. We continue applying backward saturation rules for all CDPN transition rules in Figure (7), and the union set of all generated M-tree automaton transition rules will be the transition rule set for the M-tree automaton $A'^M_{a^1,a^2}$.

From Row 2 to 12 in Figure 9 lists the key steps of applying backward saturation rules on CDPN transition rules, and they give two main clues of going backward from points a^1 and a^2 along with the $\mathcal{G}$ in Figure 6.

We start with the clue for action point a^2 with stack symbol 7, which includes automaton transition rules from (35) to (40). The transition rules from

(35) to (38) enable automaton state $q^{\natural}_{00}$ with terminal control location recognize processes with stack symbol 7 and 9. Rule (39) and (40) generated by inversing the async statement will derive two automaton states with different control locations. The one with dummy # control location has recognized a^2 action while the other one with $\natural$ control location has not recognized anything.

On the other side, the clue for point a^1 with stack symbol 2 includes automaton transition rules (41) to (43). Rules (41) and (42) enable the automaton state $q^{\#}_{10}$ recognize processes with stack symbol 12. Rule (43) requires the newest child process (spawned by finish statement) be recognized by a state with $\natural$ control location before synchronization point $wait[5, 6, 11]$ being recognized by state $q^{\#}_{\mathsf{r}10}$. It gives the semantic meaning that all children processes in the finish closure must have been terminated before the parent process execute across the synchronization point.

The M-tree automaton rules (44) and (45) will inverse the finish statement and merge the two clues given above. At this point, we can only get either the configuration of parent process be recognized by state $q^{\#}_{10}$ and child be recognized by $q^{\natural}_{\mathsf{r}00}$ or parent process be recognized by state $q^{\#}_{00}$ and child be recognized by $q^{\#}_{01}$ The child process 6(#) recognized by state $q^{\#}_{01}$ can not be merged with rule $wait[5, 6, 11](\#(Q^P * q^{\natural})) \rightarrow^* q^{\#}_{\mathsf{r}10}$ in parent process since it requires control location $\natural$ on its child's state. Hence for the initial M-configuration $\mathsf{conf}^M_{init} = 5(\#)$ can either be recognized by automaton state $q^{\#}_{10}$ or $q^{\#}_{01}$ which will not be accepted by M-tree automaton $A'^M_{a^1,a^2}$.

Appendix B: Proofs of Theorems 6 and 7

We will now prove that our type system characterizes the MHP analysis problem for statements without procedure calls.

8.1 A Characterization of CBE

Let us define $sCBE(s)$ and the helper function $runnable(s)$ as shown in Figure 10.

Lemma 1. *There exists C such that $C[a^l] = s$ if and only if $l \in runnable(s)$.*

Proof. $\leftarrow$) We must show that if $l \in runnable(s)$ then there exists C such that $C[a^l] = s$.

Let us perform induction on s and examine the seven cases.

If $s \equiv s' \,; s''$ we have two cases to consider when $l \in runnable(s)$ from Rule (65).

Suppose $s' = P'$. From this premise we must consider the cases of $l \in runnable(P')$ and $l \in runnable(s'')$.

Suppose $l \in runnable(P')$. Then from the induction hypothesis we have $C'[a^l] = P'$. We choose $C = C' \,; s''$ and since $C'[a^l] = P'$ we have $C[a^l] = s$ as desired.

$$sCBE(s_1\ ;s_2) = \begin{cases} sCBE(s_1) \cup sCBE(s_2) \cup \\ \ symcross(runnable(s_1), runnable(s_2)) & \text{if } s_1 = P \\ sCBE(s_1) & \text{otherwise} \end{cases} \quad (57)$$

$$sCBE(loop\ s) = \emptyset \quad (58)$$

$$sCBE(async\ s) = sCBE(s) \quad (59)$$

$$sCBE(finish\ s) = sCBE(s) \quad (60)$$

$$sCBE(a^l) = \emptyset \quad (61)$$

$$sCBE(skip) = \emptyset \quad (62)$$

$$sCBE(f()) = \emptyset \quad (63)$$

$$(64)$$

$$runnable(s_1\ ;s_2) = \begin{cases} runnable(s_1) \cup runnable(s_2) & \text{if } s_1 = P \\ runnable(s_1) & \text{otherwise} \end{cases} \quad (65)$$

$$runnable(loop\ s) = \emptyset \quad (66)$$

$$runnable(async\ s) = runnable(s) \quad (67)$$

$$runnable(finish\ s) = runnable(s) \quad (68)$$

$$runnable(a^l) = \{l\} \quad (69)$$

$$runnable(skip) = \emptyset \quad (70)$$

$$runnable(f()) = \emptyset \quad (71)$$

Fig. 10. Two helper functions.

Suppose $l \in runnable(s'')$. Then from the induction hypothesis we have $C''[a^l] = s''$. We choose $C = P'\ ;C''$ and since $C''[a^l] = s''$ have $C[a^l] = s$ as desired.

Suppose $s' \neq P'$. Then from this premise we have 1) $l \in runnable(s')$. Using the induction hypothesis we have that there exists C' such that 2) $C'[a^l] = s'$. We choose $C = C'\ ;s''$ and from 2) we have $C[a^l] = s$ as desired.

If $s \equiv loop\ s'$ then $runnable(s) = \emptyset$ which contradicts our premise which makes this case vacuously true.

If $s \equiv async\ s'$ then from Rule (67) we have 1) $l \in runnable(s')$. We then use the induction hypothesis with 1) to get that there exists C' such that 2) $C'[a^l] = s'$. We choose $C = async\ C'$ and with 2) we have $C[a^l] = s$ as desired.

If $s \equiv finish\ s'$ then we use similar reasoning as the previous case.

If $s \equiv a^l$ then we choose $C = \Box$ and thus have $C[a^l] = s$ as desired.

If $s \equiv skip$ then $runnable(s) = \emptyset$ which contradicts our premise which makes this case vacuously true.

If $s \equiv f()$ then $runnable(s) = \emptyset$ which contradicts our premise which makes this case vacuously true.

$\rightarrow$) We must show that if there exists C such that $C[a^l] = s$ then $l \in runnable(s)$.

Let us perform induction on s and examine the seven cases.

If $s \equiv s' \,;s''$ then there are two productions of C that conform to our premise: $C = C' \,;s''$ and $C'[a^l] = s'$ or $C = P' \,;C''$ and $C''[a^l] = s''$.

Suppose $C = C' \,;s''$ and $C'[a^l] = s'$. Then Rules (65) and (65) may apply and we must show $l \in runnable(s)$ from both rules. From this premise we use the induction hypothesis to get $l \in runnable(s')$. In either case of $s' = P'$ or $s' \neq P'$, combining this with Rule (65) give us the desired conclusion of $l \in runnable(s)$.

Suppose $C = P' \,;C''$ and $C'[a^l] = s''$. Then the first case of Rule (65) only applies to this case as $s' = P$. Using the induction hypothesis with this premise gives us $l \in runnable(s'')$. Using this with Rule (65) gives us $l \in runnable(s)$ as desired.

If $s \equiv loop\ s'$ then we see that there is no C such that $C[a^l] = s$ which contradicts our premise and makes this case vacuously true.

If $s \equiv async\ s'$ then we see that our premise is true only if there exists C' such that $C = async\ C'$ and $C'[a^l] = s'$. We use our induction hypothesis to get that $l \in runnable(s')$. Combining this with Rule (67) gives use $l \in runnable(s)$ as desired.

If $s \equiv finish\ s'$ then we use similar reasoning as the previous case.

If $s \equiv a^l$ then from Rule (69) we see our conclusion is true.

If $s \equiv skip$ then we see that there is no C such that $C[a^l] = s$ which contradicts our premise and makes this case vacuously true.

If $s \equiv f()$ then we see that there is no C such that $C[a^l] = s$ which contradicts our premise and makes this case vacuously true. □

Lemma 2. $\mathsf{CBE}(s, l_1, l_2)$ *if and only if* $(l_1, l_2) \in sCBE(s)$.

Proof. $\leftarrow$) We must show if $(l_1, l_2) \in sCBE(s)$ then $\mathsf{CBE}(s, l_1, l_2)$.

Let us perform induction on s. There are seven cases to examine.

If $s \equiv s' \,;s''$ then there are two cases from Rule (57) to consider on how $(l_1, l_2) \in sCBE(s)$.

Suppose $s' = P'$ from Rule (57). Then either $(l_1, l_2) \in sCBE(P')$, $(l_1, l_2) \in sCBE(s'')$, or
$(l_1, l_2) \in symcross(runnable(P'), runnable(s''))$.

Suppose $(l_1, l_2) \in sCBE(P')$. Then using the induction hypothesis with this premise gives us 1) $\mathsf{CBE}(P', l_1, l_2)$. From the definition of CBE we have that the exists C_1' and C_2' such that 2) $P' = C_1'[a^{l_1}]$, 3) $P' = C_2'[a^{l_2}]$ and 4) $C_1' \neq C_2'$. Let 5) $C_1 = C_1' \,;s''$ and 6) $C_2 = C_2' \,;s''$. From 4),5) and 6) we have 7) $C_1 \neq C_2$. We combine 2) with 5) and 3) with 6) to get 8) $s = C_1[a^{l_1}]$ and 9) $s = C_2[a^{l_2}]$. From the definition of CBE and 7),8) and 9) we have $\mathsf{CBE}(s, l_1, l_2)$ as desired.

Suppose $(l_1, l_2) \in sCBE(s'')$. Then using the induction hypothesis with this premise gives us 1) $\mathsf{CBE}(s'', l_1, l_2)$. From the definition of CBE we have that the exists C_1'' and C_2'' such that 2) $s'' = C_1''[a^{l_1}]$, 3) $s'' = C_2''[a^{l_2}]$ and 4) $C_1'' \neq C_2''$. Let 5) $C_1 = P' \,;C_1''$ and 6) $C_2 = P' \,;C_2''$. From 4),5) and 6) we have 7) $C_1 \neq C_2$. We combine 2) with 5) and 3) with 6) to get 8) $s = C_1[a^{l_1}]$ and 9) $s = C_2[a^{l_2}]$. From the definition of CBE and 7),8) and 9) we have $\mathsf{CBE}(s, l_1, l_2)$ as desired.

Suppose $(l_1, l_2) \in symcross(runnable(P'), runnable(s''))$. Then from the definition of $symcross()$ we have either 1) $l_1 \in runnable(P')$ and 2) $l_2 \in$

$runnable(s'')$ or vice versa. In either case we proceed using similar reasoning. Using Lemma (13) with 1) and 2) gives us that there exists C_1' and C_2' such that 3) $C_1'[a^{l_1}] = P'$ and 4) $C_2'[a^{l_2}] = s''$. Let 5) $C_1 = C_1'\ s''$ and 6) $C_2 = P'\ C_2$. From 5) and 6) we immediately see 7) $C_1 \neq C_2$. Combining 3) with 5) and 4) with 6) gives us 8) $C_1[a^{l_1}] = s$ and 9) $C_2[a^{l_2}] = s$. From the definition of CBE with 7),8) and 9) we have $\mathsf{CBE}(s, l_1, l_2)$ as desired.

Suppose $s' \neq P'$. Then we use similar reasoning as the case where $(l_1, l_2) \in sCBE(P')$.

If $s \equiv loop\ s'$ then from Rule (58) we have $sCBE(s) = \emptyset$ which contradicts our premise making this case vacuously true.

If $s \equiv async\ s'$ then from Rule (59) we have 1) $\mathsf{CBE}(s', l_1, l_2)$. From the definition of CBE we have that there exists C_1'; and C_2' such that 2) $s' = C_1'[a^{l_1}]$, 3) $s' = C_2'[a^{l_2}]$ and 4) $C_1 \neq C_2$. Let 5) $C_1 = async\ C_1'$ and 6) $C_2 = async\ C_2'$. From 4),5) and 6) we have 7) $C_1 \neq C_2$. We combine 2) with 5) and 3) with 6) to get 8) $s = C_1[a^{l_1}]$ and 9) $s = C_2[a^{l_2}]$. From the definition of CBE and 7),8) and 9) we have $\mathsf{CBE}(s, l_1, l_2)$ as desired.

If $s \equiv finish\ s'$ then we proceed using similar reasoning as the previous case.

If $s \equiv a^l$ then from Rule (61) we have $sCBE(s) = \emptyset$ which contradicts our premise making this case vacuously true.

If $s \equiv skip$ then we use similar reasoning as the previous case.

If $s \equiv f()$ then we use similar reasoning as case for a^l.

$\rightarrow$) We must show if $\mathsf{CBE}(s, l_1, l_2)$ then $(l_1, l_2) \in sCBE(s)$.

Let us perform induction on s. There are seven cases to examine.

If $s \equiv s'\ ; s''$ then from the definition of CBE we have that there exists C_1 and C_2 such that a) $s = C_1[a^{l_1}]$, b) $s = C_2[a^{l_2}]$ and c) $C_1 \neq C_2$. Let us perform case analysis on C_1 and C_2 observing that the $C\ ; s$ and $P\ ; C$ productions are the only ones that may satisfy a) and b).

Suppose $C_1 = C_1'\ ; s''$ and $C_2 = C_2'\ ; s''$. Since Rule (57) has two cases, we must consider if $s' = P'$ or $s' \neq P'$. Combining this premise with c) gives us 1) $C_1' \neq C_2'$. Additionally from this premise and a) and b) we obtain 2) $s' = C_1'[a^{l_1}]$ and 3) $s' = C_2'[a^{l_2}]$. Using the definition of CBE with 1),2) and 3) we have 4) $\mathsf{CBE}(s', l_1, l_2)$. We use the induction hypothesis with 4) to get 5) $(l_1, l_2) \in sCBE(s')$. From the Rule (57) with 5) in either case of $s' = P'$ or $s' \neq P'$ we have $(l_1, l_2) \in sCBE(s'\ ; s'')$.

Suppose $C_1 = P'\ ; C_1''$ and $C_2 = P'\ ; C_2''$. Then combining this premise with c) gives us 1) $C_1'' \neq C_2''$. Additionally from our this premise and a) and b) we obtain 2) $s'' = C_1''[a^{l_1}]$ and 3) $s'' = C_2''[a^{l_2}]$. Using the definition of CBE with 1),2) and 3) we have 4) $\mathsf{CBE}(s'', l_1, l_2)$. We use the induction hypothesis with 4) to get 5) $(l_1, l_2) \in sCBE(s'')$. From the Rule (57) combined with 5) we have $(l_1, l_2) \in sCBE(s)$ as desired.

Suppose $C_1 = C_1'\ ; s''$ and $C_2 = P'\ ; C_2''$. Combining this premise with a) and b) gives us 1) $P' = C_1'[a^{l_1}]$ and 2) $s'' = C_2''[a^{l_2}]$. From applying Lemma (1) with 1) and 2) we obtain 3) $l_1 \in runnable(P')$ and 4) $l_2 \in runnable(s'')$. From the definition of $symcross()$ we have

5) $(l_1, l_2) \in symcross(runnable(P'), runnable(s''))$. Using Rule (57) with 5) gives us $(l_1, l_2) \in sCBE(s)$ as desired.

Suppose $C_1 = P\ ; C_1''$ and $C_2 = C_2'\ ; s''$. We proceed using similar reasoning as the previous case.

If $s \equiv loop\ s'$ we see that there is no C such that $C[a^{l_1}] = s$ and thus from its definition, we have $\mathsf{CBE}(s, l_1, l_2) = false$ which contradicts our premise and making this case vacuously true.

If $s \equiv async\ s'$ then from the definition of CBE we have that there exists C_1 and C_2 such that 1) $s = C_1[a^{l_1}]$, 2) $s = C_2[a^{l_2}]$ and 3) $C_1 \neq C_2$. We observe that there is only context production that we may use with 1) and 2) to get 4) $s = (async\ C_1')[a^{l_1}]$ and 5) $s = (async\ C_2')[a^{l_2}]$. We see from 3),4) and 5) that 6) $C_1' \neq C_2'$. From 4) and 5) we may obtain 7) $s' = C_1'[a^{l_1}]$ and 8) $s' = C_2'[a^{l_2}]$. From the definition of CBE and 6),7) and 8) we have 9) $\mathsf{CBE}(s', l_1, l_2)$. Using the induction hypothesis and 9) we have 10) $(l_1, l_2) \in sCBE(s')$. Using Rule (59) we have $(l_1, l_2) \in sCBE(s)$ as desired.

If $s \equiv finish\ s'$ then we use similar reasoning as the previous case.

If $s \equiv a^l$ then we see that there is exactly one C such that $C[a^l] = s$ where $C = \Box$. From its definition then we see that $CBE(s) = false$ and contradicts our premise making this case vacuously true.

If $s \equiv skip$ then we use similar reasoning as the loop case.

If $s \equiv f()$ then we use similar reasoning as the loop case. □

Theorem 13. (CBE Characterization) $\mathsf{CBE}(s) = sCBE(s)$.

Proof. Follows from Lemma (2) and the definition of $\mathsf{CBE}()$. □

8.2 Equivalence of MHP and Types

We will give a type-based characterization of the May Happen in Parallel problem. We will show that for all statements s without procedure calls, we have $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}^{\emptyset}_{type}(s)$ (Lemma 17).

Lemma 3. *If* $B \vdash s : M, O, L$ *and* $runnable(s) \subseteq L$.

Proof. Let us perform induction on s and examine the seven cases.

If $s \equiv s_1\ ; s_2$ then from the definition of $runnable()$ we have two cases to consider from Rule (65): either $runnable(s) = runnable(s_1) \cup runnable(s_1)$ and $s = P$ or $runnable(s) = runnable(s_1)$.

Suppose $runnable(s) = runnable(s_1) \cup runnable(s_1)$ and $s = P$. From Rule (10) 1) $B \vdash s_1 : M_1, O_1, L_1$, 2) $B \vdash s_2 : M_2, O_2, L_2$ and 3) $L = L_1 \cup L_2$. Using the induction hypothesis on 1) and 2) gives us 4) $runnable(s_1) \subseteq L_1$ and 5) $runnable(s_2) \subseteq L_2$. Combining this premise with 3),4) and 5) gives us $runnable(s) \subseteq L$ as desired.

Suppose $runnable(s) = runnable(s_1)$. From Rule (10) we have 1) $B \vdash s_1 : M_1, O_1, L_1$, 2) $B \vdash s_2 : M_2, O_2, L_2$ and 3) $L = L_1 \cup L_2$. Using the induction hypothesis with this premise and 2) gives us 4) $runnable(s_1) \in L_2$. Combining this premise with 3) and 4) gives us $runnable(s) \subseteq L$ as desired.

If $s \equiv loop\ s_1$ then from the definition of $runnable()$ we have $runnable(s) = \emptyset$ which we allows us to easily see that $runnable(s) \subseteq L$.

If $s \equiv async\ s_1$ then from the definition of $runnable()$ we have that 1) $runnable(s) = runnable(s_1)$. Substituting 1) in to the premise gives us 2) $l_0 \in runnable(s_1)$. From Rule (12) we have 3) $B \vdash s_1 : M_1, O_1, L_1$ and 4) $L = L_1$. Using the induction hypothesis with 2) and 3) we obtain 5) $l_0 \in L_1$ We substitute 4) in 5) to get $l \in L$ as desired.

If $s \equiv finish\ s_1$ then we proceed using similar reasoning as the previous case.

If $s \equiv a^l$ then from the definition of $runnable()$ we have 1) $runnable(s) = \{l\}$. From Rule (14) we have 2) $L = \{l\}$ Substituting 2) in 1) gives us 3) $runnable(s) = L$. From 3) we see that $runnable(s) \subseteq L$ is true.

If $s \equiv skip$ then from the definition of $runnable()$ we have $runnable(s) = \emptyset$ which we allows us to easily see that $runnable(s) \subseteq L$.

If $s \equiv f()$ then from the definition of $runnable()$ we have $runnable(s) = \emptyset$ which we allows us to easily see that $runnable(s) \subseteq L$. □

Lemma 4. *If $B \vdash P : M, O, L$ then $runnable(P) \subseteq O$.*

Proof. Let us perform induction on P there are two cases to consider.

If $P = P_1\ ; P_2$ then from the definition of $runnable()$ we have 1) $runnable(s) = runnable(P_1) \cup runnable(P_2)$. From Rule (10) we have 2) $B \vdash P_1 : M_1, O_1, L_1$, 3) $B \vdash P_2 : M_2, O_2, L_2$ and 4) $O = O_1 \cup O_2$. Using the induction hypothesis with 2) and 3) premise gives us 5) $runnable(P_1) \subseteq O_1$ and 6) $runnable(P_2) \subseteq O_2$ Combining 1),4),5) and 6) gives us $runnable(P) \subseteq O$ as desired.

If $P = async\ s_1$ then from Rule (12) we have 1) $B \vdash s_1 : M_1, O_1, L_1$ and 2) $O = L_1$. From the definition of $runnable()$ we have 3) $runnable(P) = runnable(s_1)$. Using Lemma (3) with 1) and 4) we have 4) $runnable(s_1) \subseteq L_1$. Substituting 2) and 3) in 4) gives us $runnable(P) \subseteq O$ as desired. □

Theorem 14. *If $B \vdash s : M, O, L$, then $\mathsf{CBE}(s) \subseteq M$.*

Proof. Let us perform induction on s and examine the seven cases.

If $s \equiv s_1\ ; s_2$ then from Theorem (1) and Rule (57) we have either

$$\begin{aligned} CBE(s) = {} & CBE(s_1) \cup CBE(s_2) \cup \\ & symcross(runnable(s_1), runnable(s_2)) \end{aligned}$$

where $s_1 = P$ or $CBE(s) = CBE(s_1)$.

Suppose

$$\begin{aligned} CBE(s) = {} & CBE(s_1) \cup CBE(s_2) \cup \\ & symcross(runnable(s_1), runnable(s_2)) \end{aligned}$$

where $s_1 = P$. From Rule (10) we have 1) $B \vdash s_1 : M_1, O_1, L_1$, 2) $B \vdash s_2 : M_2, O_2, L_2$ and 3) $M = M_1 \cup M_2 \cup symcross(O_1, L_2)$. Using the induction hypothesis with 1) and 2) we obtain 4) $CBE(s_1) \subseteq M_1$ and 5) $CBE(s_2) \subseteq M_2$.

Using Lemma (4) this premise and 1) gives us 6) $runnable(s_1) \subseteq O_1$. We use Lemma (3) with 2) to get 7) $runnable(s_2) \subseteq L_2$. From 6) and 7) and the definition of $symcross()$ we have that 8) $symcross(runnable(s_1), runnable(s_2)) \subseteq symcross(O_1, L_2)$. Combining 3),4),5) and 8) we obtain $CBE(s) \subseteq M$ as desired.

Suppose $CBE(s) = CBE(s_1)$. From Rule (10) we have 1) $B \vdash s_1 : M_1, O_1, L_1$, 2) $B \vdash s_2 : M_2, O_2, L_2$ and 3) $M = M_1 \cup M_2 \cup symcross(O_1, L_2)$. Using the induction hypothesis with 1) we get 4) $CBE(s_1) \subseteq M_1$. Substituting this premise with 4) gives us 5) $CBE(s) \subseteq M_1$. Combining 3) and 5) gives us $CBE(s) \subseteq M$ as desired.

If $s \equiv loop\ s_1$ then from Theorem (1) and Rule (58) we have $CBE(s) = \emptyset$. From this we immediately may see that our conclusion is true.

If $s \equiv async\ s_1$ then from Theorem (1) and Rule (59) we have 1) $CBE(s) = CBE(s_1)$. From Rule (12) we have 2) $B \vdash s_1 : M_1, O_1, L_1$ and 3) $M = M_1$. Using the induction hypothesis with 2) we obtain 4) $CBE(s_1) \subseteq M_1$. Substituting 1) and 3) in 4) gives us $CBE(s) \subseteq M$ as desired.

If $s \equiv finish\ s_1$ then we proceed using reasoning similar to the previous case.

If $s \equiv a^l$ then we proceed using similar reasoning as the loop case.

If $s \equiv skip$ then we proceed using similar reasoning as the loop case.

If $s \equiv f()$ then we proceed using similar reasoning as the loop case.

□

The following theorem can be proved with the technique used by Lee and Palsberg [11] to prove a similar result.

Theorem 6 (Overapproximation) For a program p, a statement s in p, and a type environment B such that $\vdash p : B$, we have $\mathsf{MHP}_{sem}(s) \subseteq \mathsf{MHP}^B_{type}(s)$.

Proof. From the definition of $\mathsf{MHP}^B_{type}(s)$ we have $\mathsf{MHP}^B_{type}(s) = M$, where $B \vdash s : M, O, L$. From the definition of $\mathsf{MHP}_{sem}(s)$ we have $\mathsf{MHP}_{sem}(s) = \bigcup_{s' : s \longmapsto^* s'} \mathsf{CBE}(s')$. Suppose $s \longmapsto^* s'$. It is sufficient to show $\mathsf{CBE}(s') \subseteq M$. From Theorem 5 and induction on the number of steps in $s \longmapsto^* s'$, we have M', O', L' such that $B \vdash s' : M', O', L'$ and $M' \subseteq M$ and $O' \subseteq O$ and $L' \subseteq L$. From Theorem 14 we have $\mathsf{CBE}(s') \subseteq M'$. From $\mathsf{CBE}(s') \subseteq M'$ and $M' \subseteq M$, we conclude $\mathsf{CBE}(s') \subseteq M$. □

We now embark on proving the dual of Theorem 6 for the special case of statements without procedure calls. We begin with three lemmas that can be proved easily by induction.

Lemma 5. *If* $s_1 \longmapsto^* s_1'$ *then* $s_1 ; s_2 \longmapsto^* s_1' ; s_2$.

Proof. It is sufficient to prove this by showing if $s_1 \longmapsto^n s_1'$ then $s_1 ; s_2 \longmapsto^n s_1' ; s_2$.

We will now show that if $s_1 \longmapsto^n s_1'$ then $s_1 ; s_2 \longmapsto^n s_1' ; s_2$.

We perform induction on n. If $n = 0$ then $s_1' = s_1'$ and $s_1 \,;s_2 \longmapsto^0 s_1 \,;s_2$ is immediately obvious.

If $n = i + 1$ and we have 1) $s_1 \longmapsto s_1''$ and 2) $s_1'' \longmapsto^i s_1'$. From 1) we have that there exists a context C_1 and redex R such that 3) $s_1 = C_1[R_1]$, 4) $s_1'' = C_1[s'']$ and 5) $R_1 \to s''$. Let 6) $C = C_1 \,;s_2$. Then from the definition of $\longmapsto$ and 5) we have 7) $C[R_1] \longmapsto C[s'']$. Using 3) and 4) with 6) and 7) gives us 8) $s_1 \,;s_2 \longmapsto s_1'' \,;s_2$. Using the induction hypothesis with 2) we have 9) $s_1'' \,;s_2 \longmapsto^i s_1' \,;s_2$ which we combine with 8) to get $s_1 \,;s_2 \longmapsto^{i+1} s_1' \,;s_2$ as desired. □

Lemma 6. *If $s \longmapsto^* s'$ then async $s \longmapsto^*$ async s'.*

Proof. We use similar reasoning as Lemma (5). □

Lemma 7. *If $s \longmapsto^* s'$ then $finish\ s \longmapsto^* finish\ s'$.*

Proof. We use similar reasoning as Lemma (5). □

Lemma 8. *For all s without procedure calls, $s \longmapsto^*$ skip.*

Proof. We perform induction on s and examine the six cases. Notice that we have six cases rather than seven because s contains no procedure calls.

If $s \equiv s_1 \,;s_2$ then using the induction hypothesis we have 1) $s_1 \longmapsto^*$ *skip* and 2) $s_2 \longmapsto^*$ *skip*. Using Lemma (5) with 1) gives us 3) $s_1 \,;s_2 \longmapsto^*$ *skip* $\,;s_2$. We use $C = \Box$ and $R =$ *skip* $\,;s_2$ with Rule (2) to get 4) *skip* $\,;s_2 \longmapsto s_2$. We combine 3), 4), and 2) to get $s_1 \,;s_2 \longmapsto^*$ *skip* as desired.

If $s \equiv$ *loop* s_1 then using $C = \Box$ and $R =$ *loop* s_1 with Rule (4) we arrive at our conclusion.

If $s \equiv$ *async* s_1 then using the induction hypothesis we have 1) $s_1 \longmapsto^*$ *skip*. Using Lemma (6) with 1) yields 2) *async* $s_1 \longmapsto^*$ *async skip*. We use $C = \Box$ and $R =$ *async skip* with Rule (6) to get 3) *async skip* $\longmapsto$ *skip*. We combine 2) and 3) to obtain *async* $s_1 \longmapsto^*$ *skip* as desired.

If $s \equiv finish\ s_1$ then we proceed using similar reasoning as the previous case.

If $s \equiv a^l$ then we use $C = \Box$ and $R = a^l$ with Rule (8) to obtain our conclusion.

If $s \equiv$ *skip* the conclusion is immediate. □

Lemma 9. *If $s_2 \longmapsto^* s_2'$ then $s_1 \,;s_2 \longmapsto^* s_2'$.*

Proof. From Lemma (8) we have 1) $s_1 \longmapsto^*$ *skip*. Using Lemma (5) with 1) gives us 2) $s_1 \,;s_2 \longmapsto^*$ *skip* $\,;s_2$. Using $C = \Box$ and $R =$ *skip* $\,;s_2$ with Rule (2) we have 3) *skip* $\,;s_2 \longmapsto s_2$. Combining our premise with 2) and 3) gives us $s_1 \,;s_2 \longmapsto^* s_2'$ as desired. □

Lemma 10. *If $s \longmapsto^* s'$ then $P \,;s \longmapsto^* P \,;s'$.*

Proof. It is sufficient to prove this by showing if $s \longmapsto^n s'$ then $P\ ; s \longmapsto^n P\ ; s'$.

We will now show that if $s \longmapsto^n s'$ then $P\ ; s \longmapsto^n P\ ; s$.

We perform induction on n. If $n = 0$ then $s = s'$ and $P\ ; s \longmapsto^0 P\ ; s$ is immediately obvious.

If $n = i + 1$ and we have 1) $s \longmapsto s''$ and 2) $s'' \longmapsto^i s'$. From 1) we have that there exists a context C_1 and redex R such that 3) $s = C[R]$, 4) $s'' = C[s''']$ and 5) $R \to s'''$. Let 6) $C' = P\ ; C$. Then from the definition of $\longmapsto$ and 5) we have 7) $C'[R] \longmapsto C'[s''']$. Using 3) and 4) with 6) and 7) gives us 8) $P\ ; s \longmapsto P\ ; s''$. Using the induction hypothesis with 2) we have 9) $P\ ; s'' \longmapsto^i P\ ; s'$ which we combine with 8) to get $P\ ; s \longmapsto^{i+1} P\ ; s'$ as desired. □

Lemma 11. *If $s_1 \longmapsto^* P_1$ and $s_2 \longmapsto^* s_2'$ then $s_1\ ; s_2 \longmapsto^* P_1\ ; s_2'$.*

Proof. Using Lemma (5) with our premise gives us 1) $s_1\ ; s_2 \longmapsto^* P_1\ ; s_2$. Using the premise with Lemma (10) yields 2) $P_1\ ; s_2 \longmapsto^* P_1\ ; s_2'$. Combining 1) and 2) results in $s_1\ ; s_2 \longmapsto^* P_1\ ; s_2'$ as desired. □

Lemma 12. *If $s_1 \longmapsto^* P_1$ then $s_1\ ; s_2 \longmapsto^* P_1$.*

Proof. From Lemma (8) we have 1) $s_2 \longmapsto^* \mathit{skip}$. Using the premise and 1) with Lemma (11) gives us 2) $s_1\ ; s_2 \longmapsto^* P_1\ ; \mathit{skip}$. Using $C = \Box$ and $R = P_1\ ; \mathit{skip}$ with Rule (3) gives us 3) $P_1\ ; \mathit{skip} \longmapsto P_1$. Combining 2) and 3) results in $s_1\ ; s_2 \longmapsto^* P_1$ as desired. □

Lemma 13. *If $\emptyset \vdash s : M, O, L$ and $l \in L$ then there exists s' such that $s \longmapsto^* s'$ and $l \in \mathit{runnable}(s')$.*

Proof. Let us perform induction on s and examine the six cases. Notice that we have six cases rather than seven because $\emptyset \vdash s : M, O, L$ so s contains no procedure calls.

If $s \equiv s_1\ ; s_2$ then from Rule (10) we have a) $\emptyset \vdash s_1 : M_1, O_1, L_1$, b) $\emptyset \vdash s_2 : M_2, O_2, L_2$ and c) $L = L_1 \cup L_2$. We must consider the cases where $l \in L_1$ or $l \in L_2$.

Suppose $l \in L_1$. Then we use the induction hypothesis with a) and this premise to get that there exists s_1' such that 1) $s_1 \longmapsto^* s_1'$ and 2) $l \in \mathit{runnable}(s_1')$. Applying Lemma (5) with 1) gives us 3) $s_1\ ; s_2 \longmapsto^* s_1'\ ; s_2$. Using either Rule (65) or (65) both give 4) $l \in \mathit{runnable}(s_1'\ ; s_2)$. We choose $s' = s_1'\ ; s_2$ and from 3) and 4) we have our conclusion.

Suppose $l \in L_2$. Then we use the induction hypothesis with b) and this premise to get that there exists s_2' such that 1) $s_2 \longmapsto^* s_2'$ and 2) $l \in \mathit{runnable}(s_2')$. Using Lemma (9) with 1) gives us 3) $s_1\ ; s_2 \longmapsto^* s_2'$. We choose $s' = s_2'$ and from 2) and 3) we have our conclusion.

If $s \equiv \mathit{loop}\ s_1$ then from Rule (11) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$ and 2) $L = L_1$. Combining 2) with the premise gives us 3) $l \in L_1$. We use the induction hypothesis with 1) and 3) to get that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $l \in \mathit{runnable}(s_1')$. From Lemma (5) with 4) gives us 6) $s_1\ ; \mathit{loop}\ s_1 \longmapsto^* s_1'\ ; \mathit{loop}\ s_1$. Using $C = \Box$ and $R = \mathit{loop}\ s_1$ with Rule (5) we have 7) $\mathit{loop}\ s_1 \longmapsto$

s_1 ; *loop* s_1. Combining 6) with 7) results in 8) *loop* $s_1 \longmapsto^* s_1'$; *loop* s_1. Using Rule (65) or (65) with 5) gives us 9) $l \in runnable(s_1'$; *loop* $s_1)$. We choose $s' = s_1'$; *loop* s_1 and from 8) and 9) we reach our conclusion.

If $s \equiv async\ s_1$ then from Rule (12) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$ and 2) $L = L_1$. Combining 2) with the premise gives us 3) $l \in L_1$. We use the induction hypothesis with 1) and 3) to get that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $l \in runnable(s_1')$. From Lemma (6) and 4) we obtain 6) *async* $s_1 \longmapsto^*$ *async* s_1'. Using Rule (67) with 5) gives us 7) $l \in runnable(async\ s_1')$. We choose $s' = async\ s_1'$ and with 6) and 7) we have our conclusion.

If $s \equiv finish\ s_1$ then from Rule (13) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$ and 2) $L = L_1$. Combining 2) with the premise gives us 3) $l \in L_1$. We use the induction hypothesis with 1) and 3) to get that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $l \in runnable(s_1')$. From Lemma (7) and 4) we obtain 6) $finish\ s_1 \longmapsto^*$ $finish\ s_1'$. Using Rule (68) with 5) gives us 7) $l \in runnable(finish\ s_1')$. We choose $s' = finish\ s_1'$ and with 6) and 7) we have our conclusion.

If $s \equiv a^{l'}$ then from Rule (14) we have $L = \{l'\}$. We combine this with our premise and we have that $l = l'$. From the Rule (69) we have $runnable(s) = \{l'\}$ and since $l = l'$ we have $l \in runnable(s)$. We choose $s' = s$ and our conclusion easily follows.

If $s \equiv skip$ then from Rule (15) we have $L = \emptyset$ which contradicts our premise and makes this case vacuously true. □

Lemma 14. *If* $\emptyset \vdash s : M, O, L$ *and* $l \in O$ *then there exists* P *such that* $s \longmapsto^* P$ *and* $l \in runnable(P)$

Proof. Let us perform induction on s. There are six cases. Notice that we have six cases rather than seven because $\emptyset \vdash s : M, O, L$ so s contains no procedure calls.

If $s \equiv s_1$; s_2 then from Rule (10) we have a) $\emptyset \vdash s_1 : M_1, O_1, L_1$, b) $\emptyset \vdash s_2 : M_2, O_2, L_2$ and c) $O = O_1 \cup O_2$. We must consider the case when $l \in O_1$ or when $l \in O_2$.

Suppose $l \in O_1$. Then we use our induction hypothesis with this premise and a) to get that there exists P_1 such that 1) $s_1 \longmapsto^* P_1$ and 2) $l \in runnable(P_1)$. Applying Lemma (12) with 1) gives us 3) s_1 ; $s_2 \longmapsto^* P_1$. We choose $P = P_1$ and with 2) and 3) we have our conclusion.

Suppose $l \in O_2$. Then we use the induction hypothesis with this premise and b) to get that there exists P_2 such that 1) $s_2 \longmapsto^* P_2$ and 2) $l \in runnable(P_2)$. We use Lemma (9) with 1) to get 3) s_1 ; $s_2 \longmapsto^* P_2$. We choose $P = P_2$ and from 2) and 3) we have our conclusion.

If $s \equiv loop\ s_1$ then from Rule (11) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$ and 2) $O = O_1$. Combining 2) with our premise results in 3) $l \in O_1$. We use the induction hypothesis with 1) and 3) to get that there exists P_1 such that 4) $s_1 \longmapsto^* P_1$ and 5) $l \in runnable(P_1)$. Using Lemma (12) with 4) gives us 6) s_1 ; *loop* $s_1 \longmapsto^* P_1$. We choose $P = P_1$ and with 5) and 6) we have our conclusion.

If $s \equiv async\ s_1$ then from Rule (12) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$ and 2) $O = L_1$. Combining 2) with our premise gives us 3) $l \in L_1$. Using Lemma (13)

with 1) and 3) yields that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $l \in runnable(s_1')$. We use Lemma (6) with 4) to get 6) $async\ s_1 \longmapsto^* async\ s_1'$. Combining Rule (67) with 5) gives us 7) $l \in runnable(async\ s_1')$. We choose $P = async\ s_1'$ and from 6) and 7) we have our conclusion.

If $s \equiv finish\ s_1$ then from Rule (13) we have $O = \emptyset$ which contradicts our premise. This case is vacuously true.

If $s \equiv a^{l'}$ then we use similar reasoning as the previous case.

If $s \equiv skip$ then we use similar reasoning as the previous case. □

Lemma 15. *If $\emptyset \vdash s : M, O, L$ and $(l_1, l_2) \in M$ then there exists s' such that $s \longmapsto^* s'$ and $(l_1, l_2) \in sCBE(s')$.*

Proof. Let use perform induction on s. This gives us six cases to examine. Notice that we have six cases rather than seven because $\emptyset \vdash s : M, O, L$ so s contains no procedure calls.

If $s \equiv s_1\ ; s_2$ then from Rule (10) we have a) $\emptyset \vdash s_1 : M_1, O_1, L_1$, b) $\emptyset \vdash s_2 : M_2, O_2, L_2$ and c) $M = M_1 \cup M_2 \cup symcross(O_1, L_2)$.

Suppose $(l_1, l_2) \in M_1$. Then we may use the induction hypothesis with a) and this premise to get that there exists s_1' such that 1) $s_1 \longmapsto^* s_1'$ and 2) $(l_1, l_2) \in sCBE(s_1')$. Using 2) with Rules (57) and (57) both give us 3) $(l_1, l_2) \in sCBE(s_1'\ ; s_2)$. Using Lemma (5) with 1) we have 4) $s_1\ ; s_2 \longmapsto^* s_1'\ ; s_2$. We choose $s' = s_1'\ ; s_2$ and 3) and 4) we have our conclusion.

Suppose $(l_1, l_2) \in M_2$. Then we may use the induction hypothesis with b) and this premise to get that there exists s_2' such that 1) $s_2 \longmapsto^* s_2'$ and 2) $(l_1, l_2) \in sCBE(s_2')$. Using Lemma (9) with 1) gives us 3) $s_1\ ; s_2 \longmapsto^* s_2'$. We choose $s' = s_2'$ and from 2) and 3) we have our conclusion.

Suppose $(l_1, l_2) \in symcross(O_1, L_2)$. Then from the definition of $symcross()$ and our premise we have either 1) $l_1 \in O_1$ and 2) $l_2 \in L_2$ or vice versa. In either case, we proceed using similar reasoning. We will show for the listed cases. Using Lemma (14) with 1) and a) gives us that there exists P_1 such that 3) $s_1 \longmapsto^* P_1$ and 4) $l_1 \in runnable(P)$. Using Lemma (13) with 2) and b) gives us that there exists s_2' such that 5) $s_2 \longmapsto^* s_2'$ and 6) $l_2 \in runnable(s_2')$. From the definition of $symcross()$ with 4) and 6) we have 7) $(l_1, l_2) \in symcross(runnable(P_1), runnable(s_2'))$. From Rule (57) we have 8) $(l_1, l_2) \in sCBE(P_1\ ; s_2')$. We use Lemma (11) with 3) and 5) gives us that 9) $s_1\ ; s_2 \longmapsto^* P_1\ ; s_2'$. We choose $s' = P_1\ ; s_2'$ and from 8) and 9) we obtain our conclusion.

If $s \equiv loop\ s_1$ then from Rule (11) we have a) $\emptyset \vdash s_1 : M_1, O_1, L_1$, b) $M = M_1 \cup symcross(O_1, L_1)$ and c) $L = L_1$.

Suppose $(l_1, l_2) \in M_1$. Then using the induction hypothesis with a) and this premise gives us that there exists s_1' such that 1) $s_1 \longmapsto^* s_1'$ and 2) $(l_1, l_2) \in sCBE(s_1')$. Using 2) with Rules (57) and (57) both gives us 3) $(l_1, l_2) \in sCBE(s_1'\ ; loop\ s_1)$. We use Lemma (5) with 1) to get 4) $s_1\ ; loop\ s_1 \longmapsto^* s_1'\ ; loop\ s_1$. Using $C = \Box$ and $R = loop\ s_1$ with Rule (5) gives us 5) $loop\ s_1 \longmapsto s_1\ ; loop\ s_1$ Combining 4) and 5) gives us 6) $loop\ s_1 \longmapsto^* s_1'\ ; loop\ s_1$. We choose $s' = s_1'\ ; loop\ s_1$ and from 3) and 6) we obtain our conclusion.

Suppose $(l_1, l_2) \in symcross(O_1, L_1)$. Then from the definition of $symcross()$ and our premise we have either 1) $l_1 \in O_1$ and 2) $l_2 \in L_1$ or vice versa. In either case, we proceed using similar reasoning. Substituting c) in 2) gives us 3) $l_2 \in L$. Using Lemma (14) with a) and 1) gives us that there exists P_1 such that 4) $s_1 \longmapsto^* P_1$ and 5) $l_1 \in runnable(P_1)$. Using Lemma (13) with our original premise of $\emptyset \vdash s : M, O, L$ gives us that there exists s'' such that 6) $loop\ s \longmapsto^* s''$ and 7) $l_2 \in runnable(s'')$. From the definition of $symcross()$ with 5) and 7) we have 8) $(l_1, l_2) \in symcross(runnable(P_1), runnable(s''))$. Using Rule (57) with 8) gives us 9) $(l_1, l_2) \in sCBE(P_1\ ; s'')$. Using Lemma (11) with 4) and 6) gives us 10) $s_1\ ; loop\ s_1 \longmapsto^* P_1\ ; s''$. Using $C = \Box$ and $R = loop\ s_1$ with Rule (5) gives us 11) $loop\ s_1 \longmapsto s_1\ ; loop\ s_1$. We combine 10) and 11) to get 12) $loop\ s_1 \longmapsto^* P_1\ ; s''$. We choose $s' = P_1\ ; s''$ and from 9) and 12) we have our conclusion.

If $s \equiv async\ s_1$ then from Rule (12) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$, 2) $M = M_1$. We combine the premise with 2) to get 3) $(l_1, l_2) \in M_1$. Using the induction hypothesis with 3) we get that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $(l_1, l_2) \in sCBE(s_1')$. From Rule (59) and 5) we get 6) $(l_1, l_2) \in sCBE(async\ s_1')$. We use Lemma (6) with 4) to get 7) $async\ s_1 \longmapsto^* async\ s_1'$. We choose $s' = async\ s_1'$ and from 6) and 7) we have our conclusion.

If $s \equiv finish\ s_1$ then from Rule (13) we have 1) $\emptyset \vdash s_1 : M_1, O_1, L_1$, 2) $M = M_1$. We combine the premise with 2) to get 3) $(l_1, l_2) \in M_1$. Using the induction hypothesis with 3) we get that there exists s_1' such that 4) $s_1 \longmapsto^* s_1'$ and 5) $(l_1, l_2) \in sCBE(s_1')$. From Rule (60) and 5) we get 6) $(l_1, l_2) \in sCBE(finish\ s_1')$. We use Lemma (7) with 4) to get 7) $finish\ s_1 \longmapsto^* finish\ s_1'$. We choose $s' = finish\ s_1'$ and from 6) and 7) we have our conclusion.

If $s \equiv a^l$ then from Rule (14) we have $M = \emptyset$. This contradicts our premise and makes this case vacuously true.

If $s \equiv skip$ then we use similar reasoning as the previous case. □

Lemma 16. (Underapproximation) $\mathsf{MHP}^{\emptyset}_{type}(s) \subseteq \mathsf{MHP}_{sem}(s)$.

Proof. From Lemma (2) we have that there exists M,O and L such that $\emptyset \vdash s : M, O, L$. Using this with Lemma (15) gives us $M \subseteq \bigcup_{s':s \longmapsto^* s'} sCBE(s')$. From Theorem (13) we have $M \subseteq \bigcup_{s':s \longmapsto^* s'} CBE(s')$. From the definition of $\mathsf{MHP}^{\emptyset}_{type}(s)$ and $\mathsf{MHP}_{sem}(s) = \bigcup_{s':s \longmapsto^* s'} CBE(s')$ we have $\mathsf{MHP}^{\emptyset}_{type}(s) \subseteq \mathsf{MHP}_{sem}(s)$ as desired. □

Lemma 17. (Equivalence) *For a statement s without procedure calls, we have* $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}^{\emptyset}_{type}(s)$.

Proof. Combine Lemmas 6 and 16. □

Theorem 7. (Equivalence) For a program without recursion, where the body of main is the statement s, we have that there exists B such that $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}^{B}_{type}(s)$.

Proof. Let p be a program without recursion and let p' be a program in which a call to procedure f has been replaced with $\mathsf{body}(f)$. Additionally, let s be the body of main in p, and let s' be the body of main in p'. It is straightforward to see that 1) $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}_{sem}(s')$, and it is easy to prove by induction on s that for any type environment B, we have 2) $\mathsf{MHP}^B_{type}(s) = \mathsf{MHP}^B_{type}(s')$. We can iteratively inline each procedure call in p until we reach a program p'' without procedure calls. Let s'' be the body of main in p''. The observations 1) and 2), and the transitivity of $=$, imply that 3) $\mathsf{MHP}_{sem}(s) = \mathsf{MHP}_{sem}(s'')$, and that for any type environment B, we have 4) $\mathsf{MHP}^B_{type}(s) = \mathsf{MHP}^B_{type}(s'')$. Additionally, since p'' has no procedure calls, the type derivation for s'' does not use B, so 5) $\mathsf{MHP}^B_{type}(s'') = \mathsf{MHP}^\emptyset_{type}(s'')$. We can now use 3), Lemma 17, 5), and 4) to prove the theorem. □

Appendix C: Proof of Theorem 9

We will give the proof showing the M-tree automaton $A^M_{l,l'}$ can only recognize all M-configurations with labels (program points) l, l' that may happen in parallel.

We define a context C as a M-term in set $\mathcal{T}[X]$ with one free variable, which only appears once in the M-term. After inserting a ground M-term (M-configuration) $t = \gamma * p(t_1, \ldots, t_n)$ for $n \geq 0$, into C, we get another ground M-term $C[t]$, which also is a M-configuration.

Following the definition of ground M-term $\mathcal{T}$, M-configurations are trees in forms of $\gamma_m \ldots \gamma_1 p(t_1, \ldots, t_n)$ for $m, n \geq 0$, where $t_1, \ldots, t_n$ are sub-trees with same structure. Particularly, all leaf nodes are single p nodes without sub-trees.

To simplify the notation, we define M-tree automaton $A^M_{l,l'}[F']$ as the M-tree automaton $A^M_{l,l'}$ with terminal set being replaced with F'. It shares the same state set Q, and transition rule set δ with $A^M_{l,l'}$.

Lemma 18. *For all l, l', we have* $\mathsf{Conf}^M = L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$.

Proof. Notice that proving $\mathsf{Conf}^M = L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$ is equivalent to proving (a) $L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}]) \subseteq \mathsf{Conf}^M$, and (b) $\mathsf{Conf}^M \subseteq L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$. Because $A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}]$ is the automaton over M-terms, we always have that $L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}]) \subseteq \mathsf{Conf}^M$; proving the second condition is equal to proving for any M-configuration t we have $t \in L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$. We prove it by induction on t. There are two cases we need to analyze.

If t is the form of $p(t_1, \ldots, t_n)$ where $n \geq 0$. For $n > 0$, by induction hypothesis $t_i \in L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$, and $t_i \rightarrow^* q_{\mathsf{00}}$, we will get $p(t_1, \ldots, t_n) \rightarrow^* q_{\mathsf{p00}}$ by applying transition rule $p(Q^*, q_i, Q^*) \rightarrow q_{pi}$; for n = 0, we will directly get $p() \rightarrow q_{\mathsf{p00}}$ by transition rule. In both cases, M-configuration t is accepted by $A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}]$.

If t is the form of $\gamma(t_1)$ where t_1 is the sub-term of t. By induction hypothesis, we have $t_1 \in L(A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}])$, which gives $t_1 \rightarrow^* q_{\mathsf{00}}$. After applying transition rule $\gamma(q_i) \rightarrow q_{ri}$, we get $t = \gamma(t_1)$, $\gamma(t_1) \rightarrow^* q_{\mathsf{00}}$ and the configuration t is accepted by $A^M_{l,l'}[\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}]$. □

Lemma 19. *For all l, l', if F^{qi} is one of $\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}$, $\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}$, $\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}$, $\{q_{\mathsf{p11}}, q_{\mathsf{r11}}\}$, for $c \in \mathsf{Conf}^M$, $t \in L(A^M_{l,l'}[F^{qi}])$ and a context C such that $c = C[t]$, we have $c \in L(A^M_{l,l'}[F^{qi}])$.*

Proof. If context C is empty, and $c = t$. we have $c \in L(A^M_{l,l'}[F^{qi}])$ by equality.

If context C is not empty, we prove it by induction on M-configuration c, and there are two cases.

If c is in form of $p(t_1, \ldots, t_n)$. t is either t_j where $1 \leq j \leq n$ or sub-term of t_j. If t is t_j, $t_j \rightarrow^* q_i$ where $q_i \in F^{qi}$ by condition $t \in L(A^M_{l,l'}[F^{qi}])$, else if t is sub-term of t_j, we still have $t_j \rightarrow^* q_i$ where $q_i \in F^{qi}$ by induction hypothesis on term t_j. Based on Lemma 18, for t_k in $t_1, \ldots, t_n$ where $k \neq j$, we have $t_k \rightarrow^* q_{\mathsf{00}}$. We apply transition rule $p(Q^*, q_i, Q^*) \rightarrow q_{pi}$ on term $p(t_1, \ldots, t_n)$ with $t_j \rightarrow q_i$ and $t_k \rightarrow q_{\mathsf{00}}$, we have $p(t_1, \ldots, t_n) \rightarrow^* q_{pi}$. If $q_i \in F^{qi}$, we have $q_{pi} \in F^{qi}$ by examining the definition, thus $c \in L(A^M_{l,l'}[F^{qi}])$.

If c is in form of $\gamma(t_1)$. t is either t_1 or sub-term of t_1. If t is t_1, $t_1 \rightarrow^* q_i$ where $q_i \in F^{qi}$ by condition $t \in L(A^M_{l,l'}[F^{qi}])$, else if t is sub-term of t_1, we still have $t_1 \rightarrow^* q_i$ where $q_i \in F^{qi}$ by induction hypothesis on term t_1. We apply transition rule $\gamma(q_i) \rightarrow q_{ri}$ on term $\gamma(t_1)$ with $t_1 \rightarrow q_i$, we have $\gamma(t_1) \rightarrow^* q_{ri}$. If $q_i \in F^{qi}$, we have $q_{ri} \in F^{qi}$ by examining the definition, thus $c \in L(A^M_{l,l'}[F^{qi}])$. □

Lemma 20. *For all l, l', if F^{qi} is one of $\{q_{\mathsf{p00}}, q_{\mathsf{r00}}\}$, $\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}$, $\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}$, $\{q_{\mathsf{p11}}, q_{\mathsf{r11}}\}$, and $c \in L(A^M_{l,l'}[F^{qi}])$, then either there exists a ground term $t = p(t_1, \ldots, t_n), n \geq 0$, a context C such that $c = C[t]$ and $t \in L(A^M_{l,l'}[F^{qi}])$ but $t_1, \ldots, t_n \notin L(A^M_{l,l'}[F^{qi}])$; or there exists a ground term $t = \gamma(t')$, a context C such that $c = C[t]$ and $t \in L(A^M_{l,l'}[F^{qi}])$ but $t' \notin L(A^M_{l,l'}[F^{qi}])$.*

Proof. We prove it by induction on M-configuration c, and examine 2 cases:

If c is the term of $\gamma(t_1)$. In this case, if $t_1 \in L(A^M_{l,l'}[F^{qi}])$, based on the induction hypothesis, we can apply the lemma on t_1 and find a ground term t' and its relative context C' where $C'[t'] = t_1$ satisfying the lemma on t_1. We thus find $t = t'$ and $C = \gamma(C')$, will satisfy the lemma on c. If $t_1 \notin L(A^M_{l,l'}[F^{qi}])$ we can find a term $t = c$, $t = c \in L(A^M_{l,l'}[F^{qi}])$, but $t_1 \notin L(A^M_{l,l'}[F^{qi}])$, satisfy the lemma.

If c is the term of $p(t_1, \ldots, t_n)$ for $n \geq 0$. In this case, if there exists one t_j for $1 \leq j \leq n$ in $t_1 \ldots t_n$, that $t_j \in L(A^M_{l,l'}[F^{qi}])$, based on the induction hypothesis we can find a ground term t' and its relative context C' where $C'[t'] = t_j$ that satisfy the lemma on term t_j. We find $t = t', C = p(t_1, \ldots, C', \ldots, t_n)$, will satisfy the lemma on term c. However, if for all t_j in $t_1, \ldots, t_n$ having $t_j \notin L(A^M_{l,l'}[F^{qi}])$, we find term $t = c \in L(A^M_{l,l'}[F^{qi}])$ and all its children $t_j \notin L(A^M_{l,l'}[F^{qi}])$ satisfy the lemma. □

Lemma 21. *For all l, l', and for every M-configuration c, we have that $c \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ if and only if there exists a process in c with stack symbol l on top of the stack. Similarly, for all l, l', we have $c \in L(A^M_{l,l'}[\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}])$ if and only if there exists a process in c with stack symbol l' on top of the stack.*

Proof. Notice the two symmetric cases for tree-automaton $A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}]$ and $A^M_{l,l'}[\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}]$; so we only show the proof for $A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}]$; the other case is similar.

Proving this lemma is equal to proving the following two assertions. For any M-configuration c,

(a) if $c \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$, then there exists a process in c with stack symbol l on top of the stack
(b) if there exists a process in c with stack symbol l on the top of the stack, then $c \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$

Proof for case (a): Based on Lemma 20, we know there exists an ground M-term $t \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ and a context C where $C[t] = c$, so that either $t = p(t_1, \ldots, t_n)$, $n \geq 0$ and $t_1, \ldots, t_n \notin L(A^M_{l,l'}[F^{qi}])$, or $t = \gamma(t')$ and $t' \notin L(A^M_{l,l'}[F^{qi}])$. We apply pattern match analysis on transition rules applied on term t so that $t \to^* q_{\mathsf{10}}$, and there are 3 matched cases.

If rule $a(q_{\mathsf{p00}}) \to q_{\mathsf{r10}}$ is matched. In this case, we have t is in form of $\gamma(t')$, $\gamma = l$, $t' \to^* q_{\mathsf{p00}}$. Again, we apply pattern match on $t' \to^* q_{\mathsf{p00}}$, we have 2 cases matched. If $p(Q^*, q_i, Q^*) \to q_{p_i}$ matched, we have $t' \to^* p(Q^*, q_{\mathsf{00}}, Q^*)$, and $\gamma(t') \to^* l(p(Q^*, q_{\mathsf{00}}, Q^*))$ that give stack symbol l on top of the control location p. Else if $p() \to q_{\mathsf{p00}}$ is matched, we get $t' = p()$, and $\gamma(t') = l(p())$, which has symbol l on top of the control location p.

If rule $\gamma(q_i) \to q_{ri}$ is matched. In this case, we have t in the form of $\gamma(t')$, $t' \to^* q_{\mathsf{10}}$. Because we have $t' \notin L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ in the hypothesis, this rule match failed.

If rule $p(Q^*, q_i, Q^*) \to q_{pi}$ is matched. In this case, we have t is in the form of $p(t_1, \ldots, t_n)$ where $t_j \to^* q_{\mathsf{10}}$ for $t_j \in t_1 \ldots t_n$. Because we have $t_j \notin L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ in the hypothesis, this rule match failed.

All matched cases above indicate there exists a process in c with stack symbol l on top of stack.

Proof for case (b): In this case we have a ground term t which is a process in form of $\gamma p(t_1, \ldots, t_n)$ where $n \geq 0$, and $\gamma = l$. If $n > 0$, based on Lemma 18, we have all $t_j \to^* q_{\mathsf{00}}$ for $t_j \in t_1 \ldots t_n$, and $p(t_1, \ldots, t_n) \to^* q_{\mathsf{p00}}$ by applying transition rule $p(Q^*, q_i, Q^*) \to q_{pi}$. Else if $n = 0$, we have $p(t_1, \ldots, t_n) \to^* q_{\mathsf{p00}}$ by applying transition rule $p() \to q_{\mathsf{p00}}$, then we get $\gamma p(t_1, \ldots, t_n) \to^* q_{\mathsf{10}}$ by applying transition rule $a(q_{\mathsf{p00}}) \to q_{\mathsf{10}}$. We can find a context C where $C[t] = c$, based on Lemma 19, so that $c = C[t] \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$. □

Theorem 9. $\mathsf{Conf}^M_{l,l'} = L(A^M_{l,l'})$.

Proof. To prove this theorem, we show

(a) For any M-configuration $c \in L(A^M_{l,l'})$, we have $c \in \mathsf{Conf}^M_{l,l'}$;
(b) For any M-configuration $c \in \mathsf{Conf}^M_{l,l'}$, we have $c \in L(A^M_{l,l'})$.

Proof for (a): In this case, we have $c \in L(A^M_{l,l'})$, and based on Lemma 20 we can find a ground M-term $t \in L(A^M_{l,l'})$ and a context C where $C[t] = c$, so

that either $t = p(t_1, \ldots, t_n)$, $n \geq 0$ and $t_1, \ldots, t_n \notin L(A^M_{l,l'})$, or $t = \gamma(t')$ and $t' \notin L(A^M_{l,l'})$. We apply pattern match analysis on transition rules applied on term t so that $t \to^* q_{11}$ there are 6 matched cases.

If rule $l(q_{\mathsf{p01}}) \to q_{\mathsf{r11}}$ is matched, we have t in form of $\gamma(t')$, $\gamma = l$ and $t' \to^* q_{\mathsf{p01}}$. Again we apply pattern match on $t' \to^* q_{\mathsf{p01}}$, and there is 1 matched rule $p(Q^*, q_i, Q^*) \to q_{pi}$. We have $t' = p(t_1, \ldots, t_n)$, where $t_i \to^* q_{\mathsf{01}}$ for $t_i \in t_1 \ldots t_n$. Then we get term $\gamma(t')$ has a stack symbol l on top of a control location p, which indicates stack symbol l on the top of stack in a process with control location p. Because sub-term t_i is recognized by M-tree automaton $A^M_{l,l'}[\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}]$, based on Lemma 21, we have another process in sub term t_i with symbol l' on the top of the stack.

If rule $l'(q_{\mathsf{p10}}) \to q_{\mathsf{r11}}$ is matched, we get the same result as previous case, since they are symmetric.

If rule $p(Q^*, q_{\mathsf{10}}, Q^*, q_{\mathsf{01}}, Q^*) \to q_{\mathsf{p11}}$ is matched, we have sub term t_i, t_j where $i \neq j$ that $t_i \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ and $t_j \in L(A^M_{l,l'}[\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}])$. Based on Lemma 21, we will get two processes in subterms t_i and t_j with symbol l, and symbol l' on top of the stacks respectively.

If rule $p(Q^*, q_{\mathsf{01}}, Q^*, q_{\mathsf{10}}, Q^*) \to q_{\mathsf{p11}}$ is matched, it gives the same result as previous rule.

If rule $\gamma(q_i) \to q_{ri}$ is matched, $t = \gamma(t')$, where $t' \to^* q_{\mathsf{11}}$. This is conflicting with condition that $t' \notin L(A^M_{l,l'})$, so the rule match fails.

If rule $p(Q^*, q_i, Q^*) \to q_{pi}$ is matched, $t = p(t_1, \ldots, t_n)$, where $t_j \to^* q_{\mathsf{11}}$ for $t_j \in t_1 \ldots t_n$. This is conflicting with condition that $t_j \notin L(A^M_{l,l'})$, so the rule match failed.

All matched patterns above identify ground M-term t that has two processes with stack symbol l and l' on top of their stacks respectively, so that configuration $c = C[t]$ belongs to the family of $\mathsf{Conf}^M_{l,l'}$.

Proof for (b): in this case, for any M-configuration c in family $\mathsf{Conf}^M_{l,l'}$, there will be 2 processes with stack symbol l and l' on top of stacks. Thus, we get two ground M-terms $t_l = l(p(t_1, \ldots, t_n)), n \geq 0$, $t_{l'} = l'(p(t'_1, \ldots, t'_m)), m \geq 0$. We apply transition rules $p(Q^*, q_i, Q^*) \to q_{pi}$, $l(q_{\mathsf{p00}}) \to q_{\mathsf{r10}}$ and $l'(q_{\mathsf{p00}}) \to q_{\mathsf{r01}}$ onto t_l and $t_{l'}$ with $t_i \to^* q_{\mathsf{00}}$ and $t'_i \to^* q_{\mathsf{00}}$ for $t_i \in t_1 \ldots t_n$, $t'_i \in t'_1 \ldots t'_m$ given by Lemma 18, then get $t_l \to^* q_{\mathsf{10}}$, $t_{l'} \to^* q_{\mathsf{01}}$. Because M-configuration c is a tree, we do case analysis on positions of M-terms t_l and $t_{l'}$ on the tree. There are 3 cases.

If t_l is in one of the sub-term t'_i in $t_{l'}$. In this case, based on Lemma 21, we will get $t'_i \to^* q_{\mathsf{10}}$, since there is a process in t'_i with symbol l on top of the stack. Next we apply transition rule $p(Q^*, q_i, Q^*) \to q_{pi}$ and get $p(t'_1, \ldots, t'_n) \to^* q_{\mathsf{p10}}$. Finally apply rule $l'(q_{\mathsf{p10}}) \to q_{\mathsf{r11}}$, we get $t_{l'} = l'(p(t'_1, \ldots, t'_n)) \to^* q_{\mathsf{11}}$. Because we can find a context C where $C[t_{l'}] = c$, based on Lemma 19 we have $c \in L(A^M_{l,l'})$ since $t_{l'} \in L(A^M_{l,l'})$.

If $t_{l'}$ is in one of the sub-term t_i in t_a. In this symmetric case, we get the same result as previous proof.

If t_l and $t_{l'}$ are two separator terms on the tree. In this case, they must have a closest common ancestor term $t_c = q(t_1, \ldots, t_i, \ldots, t_j, \ldots, t_n)$ where t_l is in

the sub term of t_i and $t_{l'}$is in the sub term of t_j (or the symmetric case that $t_{l'}$ is in the sub term of t_i and t_l is in the sub term of t_j) . Based on Lemma 21, we have $t_i \in L(A^M_{l,l'}[\{q_{\mathsf{p10}}, q_{\mathsf{r10}}\}])$ and $t_j \in L(A^M_{l,l'}[\{q_{\mathsf{p01}}, q_{\mathsf{r01}}\}])$ which is equivalent to $t_i \to^* q_{\mathsf{10}}$, $t_j \to^* q_{\mathsf{01}}$. By applying transition rule $p(Q^*, q_{\mathsf{10}}, Q^*, q_{\mathsf{01}}, Q^*) \to q_{\mathsf{p11}}$, (or $p(Q^*, q_{\mathsf{01}}, Q^*, q_{\mathsf{10}}, Q^*) \to q_{\mathsf{p11}}$ in the symmetric case) we derive that we have $p(t_1, \ldots, t_i, \ldots, t_j, \ldots, t_n) \to^* q_{\mathsf{p11}}$, and term t_c is recognized by $A^M_{l,l'}$. We can find a context C for term t_c so that $C[t_c] = c$, then by Lemma 19 we will get $c \in L(A^M_{l,l'})$, since $t_c \in L(A^M_{l,l'})$.

All cases above give the result that $c \in L(A^M_{l,l'})$. □