Action-Level Addition of Leads-To Properties to Shared Memory Parallel Programs

Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-08-01
March 2008

Action-Level Addition of Leads-To Properties to Shared Memory Parallel Programs

Ali Ebnenasir

March 2008

Abstract

We present a method for large-scale automatic addition of Leads-To properties to shared memory parallel programs. Such an automated addition of Leads-To is highly desirable in facilitating the design of multicore programs for average programmers. Our approach is *action-level* in that we separately analyze and revise the actions of an existing program in order to accommodate a new Leads-To property while preserving already existing properties. Based on our method, we have developed a software framework that exploits the computational resources of geographically distributed machines to add Leads-To properties to parallel programs. We demonstrate our approach in the context of a token passing and a barrier synchronization program.

Contents

1 Introduction

The use of multicore processors in commodity systems brings parallelism to mainstream programmers. To facilitate the design of multicore programs, we need automated techniques as manual design of parallel programs is a difficult task in part due to the inherent non-determinism of such programs. The design of parallel programs becomes even more complicated when developers are given new requirements (specified as new desired properties) after they already have created some design artifacts. In such scenarios, designers are left with two options: either design a whole new program from scratch, or *add* the new properties to the existing design (i.e., *revise* the existing design) while preserving existing properties. The focus of this paper is on the latter as we conjecture that design revisions will potentially be less expensive. Such an addition of new properties is especially difficult for liveness properties (e.g., guaranteed service) [1]. We present a method for large-scale addition of the commonly-used Leads-To property to shared memory parallel programs, where a Leads-To property states that the program will eventually meet some condition (e.g., will enter its critical section) once another condition holds (e.g., upon trying to enter the critical section).

Numerous approaches exist for automated design and verification of parallel programs [2–8] most of which focus on designing/verifying an *integrated* combinatorial model of a program (implemented as a reachability graph [6–8] or as a set of Binary Decision Diagrams (BDDs) [9] in symbolic methods [4]) which makes it difficult to scale up such methods as the size[1] of the program increases. For example, specification-based approaches [2, 5] synthesize the abstract structure of a parallel program as a finite-state machine from the satisfiability proof of its temporal logic specification. Model checking techniques [3, 10, 11] verify the correctness of a program model with respect to temporal logic specifications. Compositional design and verification methods [12, 13] provide a divide-and-conquer approach for reasoning about the correctness of concurrent programs. Our previous work [1] focuses on adding Leads-To properties to integrated models of parallel programs. While all aforementioned approaches are important and useful for automated design of parallel programs, we believe that improving the scalability of automated design is equally important if we want to facilitate the design of parallel programs for average programmers.

We present an action-level method for dividing the problem of adding liveness (e.g., Leads-To) properties to parallel programs. Our focus on liveness properties is due to the fact that adding safety has been extensively investigated [14–16], and adding liveness in a way that easily scales up remains a challenge. In our approach, we start with an existing program p and a new liveness requirement $\mathcal{L}$ that is not met by p. Then we investigate the following questions: *Is there a revised version of p that satisfies $\mathcal{L}$ while preserving its existing properties*? *How can we answer the above question by analyzing each instruction (action) of p separately*? *If the answer to the first question is affirmative, then which program actions should be changed so that p satisfies $\mathcal{L}$ (while preserving its existing properties)*?

The proposed approach in this paper investigates the above questions for shared memory parallel programs. Specifically, in our model, programs are defined in terms of processes, where each process has a finite set of variables and a finite set of Dijkstra's guarded commands [17], called *actions*, that can be executed non-deterministically. A guarded command is of the form $grd \rightarrow stmt$, where grd is a Boolean expression specified over program variables and *stmt atomically* updates a subset of program variables when grd holds. (Such a design language can be realized in several existing compilers (e.g., [18]) that support software transactional memory [19].) The input to our method is a parallel program that satisfies a set of Propositional Linear Temporal Logic (PLTL) [20] properties, denoted *Spec*. We also take a new Leads-To property of the form $\mathcal{L} \equiv P \leadsto Q$, where $P \leadsto Q$ means that *it is always the case that if P holds in a program execution then Q will eventually hold at some point of time in that execution.* Subsequently, we distribute each program action over a network of workstations. Then we analyze and possibly revise each action on a different machine in order to synthesize a revised version of the input program that satisfies ($Spec \wedge \mathcal{L}$). Such an analysis is based on a distributed synthesis algorithm that has two layers: the *ranking* layer and the *synthesis* layer. The ranking layer assigns a rank to all reachable states based on their shortest distance to any state in Q. The synthesis layer uses the results of ranking to ensure reachability of Q from every state in P. Both layers have one component associated with each program action. The ranking and synthesis components of different actions can be deployed on different machines to enhance the space efficiency of addition. The hypothesis of

[1]In this work, we determine the size of the program by the number of its processes, which has a direct relation with the total number of program actions, variables, and state space.

this work is that an action-level addition of Leads-To properties improves the scalability of addition.

Based on our action-level method, we have developed a software framework, called PARallel program DESigner (PARDES), that is deployed on a cluster of machines and adds Leads-To properties to shared memory parallel programs. Using PARDES, we have designed several parallel programs among which (i) a token passing program, and (ii) a barrier synchronization protocol. The organization of the paper is as follows: We present preliminary concepts in Section 2. In Section 3, we represent the problem of adding Leads-To properties from [1]. Subsequently, in Section 4, we present an overview of our proposed approach. Then, in Section 5, we present our action-level method. In Section 6, we demonstrate our approach in the context of a barrier synchronization protocol and a mutual exclusion program. In Section 7, we illustrate how our approach made it possible to easily scale up the addition of Leads-To to a barrier synchronization program with 18 processes on a cluster of 5 distributed machines. We discuss related work in Section 8. Finally, we make concluding remarks and discuss future work in Section 9.

2 Preliminaries

In this section, we present formal definitions of programs and specifications. The definition of specification is adapted from Alpern and Schneider [21] and Emerson [20]. We represent our read/write model from [6,22]. To illustrate our modeling approach, we use a Token Passing (TP) program as a running example and use sans serif font for the exposition of the TP example.

Programs and processes. A *program* $p = \langle V_p, \Pi_p \rangle$ is a tuple of a finite set V_p of variables and a finite set Π_p of processes. Each variable $v_i \in V_p$, for $1 \leq i \leq n$, has a finite non-empty domain D_i. A *state* s of a program p is a valuation $\langle d_1, d_2, \cdots, d_n \rangle$ of program variables $\langle v_1, v_2, \cdots, v_n \rangle$, where d_i is a value in D_i. We denote the value of a variable v_i in a state s by $v_i(s)$. The *state space* $\mathcal{Q}_p$ is the set of all possible states of p. A *transition* of p is of the form (s, s'), where s and s' are program states. A *process* P_j , $1 \leq j \leq k$, includes a finite set of *actions*. An action is a guarded command (due to Dijkstra [17]) of the form $grd \rightarrow stmt$ that captures the set of transitions (s_0, s_1) where the Boolean expression grd (specified over V_p) holds in s_0 and the *atomic* execution of the assignment $stmt$ changes the program state to s_1. An assignment updates a subset of variables in V_p and always terminates once executed. The set of program actions is the union of the actions of all its processes. We represent the new values of updated variables as *primed* values. For example, if an action updates the value of an integer variable v_1 from 0 to 1, then we have $v_1 = 0$ and $v_1' = 1$.

TP example. We consider a Token Passing (TP) program that has three processes P_1, P_2 and P_3. Each process P_i $(1 \leq i \leq 3)$ has a integer variable x_i with the domain {0, 1, 2, 3}. We denote a program state by a tuple $\langle x_1, x_2, x_3 \rangle$. Process P_i, for $i = 2, 3$, copies the value of x_{i-1} to x_i without any preconditions. We present the actions of P_2 and P_3 as the following parameterized action:

$$A_i: \quad true \quad \longrightarrow \quad x_i := x_{i-1};$$

Process P_1 copies the value $(x_3 \oplus 1)$ to x_1 (without any preconditions), where $\oplus$ denotes addition in modulo 4. The action of P_1 is as follows:

$$A_1: \quad true \quad \longrightarrow \quad x_1 := x_3 \oplus 1;$$

By definition, process P_i, $i = 2, 3$, has a token iff $x_i \neq x_{i-1}$. Process P_1 has a token iff $x_1 = x_3$. The above actions represent a translation of the following piece of code specified in three different threads in the Intel C++ language extension for Software Transactional Memory, where __tm_atomic declares a block of statements that are executed atomically:

```
int x1, x2, x3;
void* thread_1(void* arg)
{
__tm_atomic {
      x1 = (x3+1) % 4;
    }
}

void* thread_2(void* arg)
```

```
{
__tm_atomic {
      x2 = x1;
    }
}

void* thread_3(void* arg)
{
__tm_atomic {
      x3 = x2;
    }
}
```

State and transition predicates. A *state predicate* of p is a subset of $\mathcal{Q}_p$ specified as a Boolean expression over V_p.[2] An *unprimed* state predicate is specified only in terms of unprimed variables. Likewise, a *primed* state predicate includes only primed variables. A *transition predicate* (adapted from [22, 23]) is a subset of $\mathcal{Q}_p \times \mathcal{Q}_p$ represented as a Boolean expression over both unprimed and primed variables. We say a state predicate X *holds at a state* s (respectively, $s \in X$) iff (if and only if) X evaluates to true at s. (A state predicate X also represents a transition predicate that includes all transitions (s, s'), where either $s \in X$ and s' is an arbitrary state, or s is an arbitrary state and $s' \in X$.) An action $grd \rightarrow stmt$ is enabled at a state s iff grd holds at s. A process $P_j \in \Pi_p$ is enabled at s iff there exists an action of P_j that is enabled at s. We define a function *Primed* (respectively, *UnPrimed*) that takes a unprimed (respectively, primed) state predicate X (respectively, X') and substitutes each variable in X (respectively, X') with its primed (respectively, unprimed) version, thereby returning a state predicate X' (respectively, X). The function *extractPrimed* (respectively, *extractUnPrimed*) takes a transition predicate T and returns a primed (respectively, an unprimed) state predicate representing the set of destination (respectively, source) states of all transitions in T.

TP example. We define a state predicate $\mathcal{S}_1$ that captures the set of states in which only one token exists in the TP program, where

$$\begin{aligned}\mathcal{S}_1 = \; & ((x_1 = x_2) \wedge (x_2 = x_3)) \vee \\ & ((x_1 = x_2) \wedge (x_2 \neq x_3)) \vee \\ & ((x_1 \neq x_2) \wedge (x_2 = x_3))\end{aligned}$$

For example, the states $s_1 = \langle 0, 1, 1 \rangle$ and $s_2 = \langle 2, 2, 3 \rangle$ belong to $\mathcal{S}_1$, where P_2 has a token in s_1 and P_3 has a token in s_2. As another example, we consider a subset of $\mathcal{S}_1$ where all x values are equal. Each state in $\mathcal{S}_2$ is called a *stable* state [24]. Formally, we have

$$\mathcal{S}_2 = ((x_1 = x_2) \wedge (x_2 = x_3))$$

Closure. A state predicate X is *closed in an action $grd \rightarrow stmt$* iff executing $stmt$ from a state $s \in (X \wedge grd)$ results in a state in X. We say a state predicate X is *closed in a program p* iff X is closed in all actions of p [25].

TP example. The state predicates $\mathcal{S}_1$ is closed in actions A_1 and A_2. We leave it to the reader to verify this claim.

Program computations and execution semantics. We consider a nondeterministic interleaving of all program actions generating a sequence of states. A *computation* of a program $p = \langle V_p, \Pi_p \rangle$ is an infinite sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}), where $i \geq 0$, in σ, there exists an action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$ such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}, i.e., action $grd \rightarrow stmt$ *induces* the transition (s_i, s_{i+1}); (2) σ is *maximal* in that either σ is infinite or if it is finite and its final state is s_f, then no program action induces a transitions (s_f, s), where $s \neq s_f$, and (3) σ is *fair*; i.e., if a process P_j is continuously enabled in σ, then eventually some action of P_j will be executed. A *computation prefix* of p is a finite sequence $\sigma = \langle s_0, s_1, \cdots, s_n \rangle$ of states in which every transition (s_i, s_{i+1}), for $0 \leq i < n$, is induced by some action of some process in Π_p. Since all program computations are infinite, during the addition of Leads-To, we distinguish between a valid *terminating computation* and a *deadlocked computation* by stuttering at the final state of valid terminating computations. However, if a program computation σ reaches a state s with no outgoing transitions, then s is a deadlocked state and σ is a deadlocked (i.e., *invalid*) computation.

Read/write model. In our model, processes are allowed to read all program variables in an atomic step. However, we specify write restrictions that determine the set of variables that are writeable for each

[2] An individual state s, the empty set, and the entire state space (i.e., the universal set) are special cases of a state predicate.

process P_j, denoted w_j. No action in a process P_j is allowed to update a variable $v \notin w_j$. In addition to the write restrictions for each process, we allow additional write constraints on actions in that a variable that is writeable for a process may not be writeable for a specific action of that process. The overall write restrictions for each action of a process P_j can be specified as a transition predicate $wRest \equiv (\forall v : v \notin w_j : v = v')$; i.e., the value of an unprimed variable $v \notin w_j$ should be equal to the value of its primed version. Using the write restrictions $wRest$, we formally specify an action $grd \rightarrow stmt$ as a transition predicate $grd \wedge stmtExpr \wedge wRest$, where $stmtExpr$ is a Boolean expression generated from the assignment $stmt$. For example, an assignment $x := 1$ can be specified as the primed expression $x' = 1$.

TP example. Each process P_i is allowed to write only x_i ($1 \leq i \leq 3$). As an example of a transition predicate, we represent the action A_2 as $(x_2' = x_1') \wedge wRest$, where $wRest$ is the transition predicate $(x_1 = x_1') \wedge (x_3 = x_3')$.

Specifications. We recall the syntax and semantics of PLTL from Emerson [20]. The following rules generate the set of PLTL formulas: (1) each atomic proposition, specified as a Boolean expression over program variables, is a formula; (2) if $\mathcal{F}_1$ and $\mathcal{F}_2$ are two PLTL formulas, then $\mathcal{F}_1 \wedge \mathcal{F}_2$ and $\neg\mathcal{F}_1$ are PLTL formulas as well, and (3) if $\mathcal{F}_1$ and $\mathcal{F}_2$ are two PLTL formulas, then $\mathcal{F}_1 U \mathcal{F}_2$ (read it as $\mathcal{F}_1$ is true *until* $\mathcal{F}_2$ becomes true) and $X\mathcal{F}_1$ (read it as in the *next* state $\mathcal{F}_1$ is true) are PLTL formulas. We define the semantics of PLTL formulas with respect to infinite sequences of state $\sigma = \langle s_0, s_1, \cdots \rangle$. A sequence σ *models* an atomic proposition $\mathcal{AP}$ (denoted $\sigma \models \mathcal{AP}$) iff $\mathcal{AP}(s_0)$ evaluates to true, where $\mathcal{AP}(s_0)$ denotes the expression resulting from substituting all program variables in $\mathcal{AP}$ with their values in state s_0. We say σ *models* $(\mathcal{F}_1 \wedge \mathcal{F}_2)$, denoted $\sigma \models (\mathcal{F}_1 \wedge \mathcal{F}_2)$, iff $\sigma \models \mathcal{F}_1$ and $\sigma \models \mathcal{F}_2$. Moreover, $\sigma \models \neg\mathcal{F}_1$ iff it is not the case that $\sigma \models \mathcal{F}_1$. For a sequence σ, we have $\sigma \models (\mathcal{F}_1 U \mathcal{F}_2)$ iff there exists a state s_j, $j \geq 0$, in σ, where $\mathcal{F}_2$ holds at s_j, and for every s_k, where $k < j$, $\mathcal{F}_1$ holds at s_k. Due to practical significance, we define the eventuality $\sigma \models \Diamond\mathcal{F}$ that means eventually $\mathcal{F}$ becomes true in the sequence σ (i.e., $\Diamond\mathcal{F} \equiv (true U \mathcal{F})$). Further, we define $\sigma \models \Box\mathcal{F}$, where $\Box\mathcal{F}$ means formula $\mathcal{F}$ is *always* true in every state of the sequence σ (i.e., $\Box\mathcal{F} \equiv \neg\Diamond(\neg\mathcal{F})$).

A *specification*, denoted *Spec*, is indeed a set of infinite sequences of states $\sigma = \langle s_0, s_1, \cdots \rangle$ such that each element σ in *Spec* models all formulas in a given (finite) set of PLTL formulas; i.e., $\sigma \models \wedge_{i=1}^{k} \mathcal{F}_i$, where k is a fixed positive integer and $\mathcal{F}_i$ is a PLTL formula. We say a program *computation* $\sigma = \langle s_0, s_1, \cdots \rangle$ *satisfies (i.e., does not violate) Spec from a state predicate* Y iff $s_0 \in Y$ and σ is in *Spec*. A *program p satisfies Spec from a state predicate* Y iff every computation of p satisfies *Spec* from Y.

Leads-To property. For the state predicates P and Q, a *computation* $\sigma = \langle s_0, s_1, \cdots \rangle$ *satisfies the* Leads-To *property* $P \leadsto Q$ from a state predicate Y iff we have $s_0 \in Y$, and for every state s_i, $i \geq 0$, if P is true in s_i, then there exists a state s_j, $j \geq i$, where Q holds at s_j. A *program p satisfies* $P \leadsto Q$ *from a state predicate* Y iff every computation of p satisfies $P \leadsto Q$ from Y. ($P \leadsto Q$ is logically equivalent to $(\Box(P \Rightarrow \Diamond Q))$.)

TP example. Suppose that the designer of the TP program starts with a simple program with three concurrent processes, where each process P_i includes the action A_i ($1 \leq i \leq 3$). The ultimate goal of the designer is to develop a program that starting from any arbitrary state will eventually reach a stable state, where all x values are the same (i.e., all processes synchronize on the same value). Formally, the TP program should satisfy ($true \leadsto \mathcal{S}_2$). To achieve this goal, the designer decomposes this problem into two subproblems. First, the designer creates an intermediate program TP' that, from any arbitrary state, reaches a state where only one process has a token; i.e., add the property $\mathcal{L}_1 \equiv (true \leadsto \mathcal{S}_1)$ to TP. Second, the designer generates a revised version of TP', denoted TP", that, starting from any state where only one process has a token, will reach a stable state; i.e., add $\mathcal{L}_2 \equiv (\mathcal{S}_1 \leadsto \mathcal{S}_2)$ to TP' while preserving the property $\mathcal{L}_1$.

Invariants. A state predicate $\mathcal{I}$ is an invariant of a program p for its specification *Spec* iff the following conditions hold: (1) $\mathcal{I}$ is closed in p, and (2) p satisfies *Spec* from $\mathcal{I}$.

TP example. The initial invariant of the TP program is equal to its entire state space that includes 4^3 states.

3 Problem Statement

In this section, we formulate the problem of adding Leads-To properties to shared memory parallel programs. Consider a Leads-To property $\mathcal{L} \equiv P \leadsto Q$ for state predicates P and Q. Let $p = \langle V_p, \Pi_p \rangle$ be a program that satisfies its specification *Spec* from its invariant $\mathcal{I}$, but violates $\mathcal{L}$ from $\mathcal{I}$. We wish to generate a revised version of p, denoted $p^r = \langle V_p, \Pi_p^r \rangle$,[3] with a new invariant $\mathcal{I}^r$ such that p^r satisfies $(Spec \wedge \mathcal{L})$ from $\mathcal{I}^r$. More precisely, p^r should satisfy $\mathcal{L}$ from $\mathcal{I}^r$ while preserving *Spec*. Satisfying *Spec* from $\mathcal{I}^r$ means that p^r should have infinite computations starting from any state in $\mathcal{I}^r$; i.e., p^r should not deadlock starting from any state

[3]Note that the set of variables of p^r is the same as the set of variables of p; i.e., no new variables are added while revising p.

in $\mathcal{I}^r$. Moreover, while generating p^r, we want to benefit from the assumption that p satisfies *Spec* from $\mathcal{I}$. In other words, we would like to reuse the computations of p generated from $\mathcal{I}$; i.e., reuse the correctness of p from $\mathcal{I}$.

If $\mathcal{I}^r$ includes a state s that does not belong to $\mathcal{I}$, then the execution of p from s may generate new computations, which makes it difficult to reuse the computations of p from $\mathcal{I}$. Hence, we require that $\mathcal{I}^r \Rightarrow \mathcal{I}$. Moreover, starting in $\mathcal{I}^r$, the actions of p^r should not include new transitions. Otherwise, p^r may exhibit new behaviors. Thus, for each action $grd^r \rightarrow stmt^r$ in p^r, we require that there exists an action $grd \rightarrow stmt$ in p such that the transition predicate $\mathcal{I}^r \wedge grd^r \wedge stmt^r Expr \wedge wRest$ implies the transition predicate $\mathcal{I}^r \wedge grd \wedge stmtExpr \wedge wRest$. Therefore, we formally state the problem of adding Leads-To as follows:

The Problem of Adding Leads-To.
Given p, $\mathcal{I}$, *Spec*, and $\mathcal{L} \equiv P \leadsto Q$, identify p^r and $\mathcal{I}^r$ such that the following conditions hold:
(1) $\mathcal{I}^r \Rightarrow \mathcal{I}$,
(2) For each action $grd^r \rightarrow stmt^r$ in p^r, there exists some action $grd \rightarrow stmt$ in p such that
$(\mathcal{I}^r \wedge grd^r \wedge stmt^r Expr \wedge wRest) \Rightarrow$
$(\mathcal{I}^r \wedge grd \wedge (stmtExpr \wedge wRest)$,
where $wRest$ captures the write restrictions of the action $grd \rightarrow stmt$.
(3) p^r satisfies $(Spec \wedge \mathcal{L})$ from $\mathcal{I}^r$. □

Remark. Note that we have not explicitly stated the deadlock-freedom requirement since it follows from 'p^r satisfies *Spec*'. During the addition of Leads-To properties, we ensure that program computations do not deadlock.

4 Overview of the Action-Level Addition

In this section, we present an overview of our approach (Figure 1) for action-level addition of Leads-To properties. To add $P \leadsto Q$, we must ensure that any program computation that reaches a state in P will eventually reach a state in Q. In our previous work [1], we provide such an assurance by exhaustive analysis and revision of integrated models of parallel programs. Our analysis in [1] includes an exhaustive search for (1) cycles reachable from P whose all states are outside Q, and (2) states reachable from P from where Q cannot be reached. To resolve cycles, we use a ranking process that assigns a rank to each state s, where $Rank(s)$ is the length of the shortest computation prefix from s to any state in Q. Afterwards, we eliminate transitions (s_0, s_1) such that (1) s_0 is reachable from P and s_0 does not belong to Q and $Rank(s_0) \leq Rank(s_1)$; (2) $Rank(s_1) = \infty$, i.e., there are no computation prefixes from s_1 that reach Q, or (3) s_1 becomes a deadlock state due to removing some transitions in cases (1) and (2). The algorithm presented in [1] adds Leads-To to an integrated model of a parallel program which may be represented as a reachability graph. A limitation of the work in [1] is the state space explosion problem. One way to mitigate this problem is to follow the approach used in parallel/distributed model checking [26, 27] where the set of reachable states is distributed over a network of workstations. However, such techniques suffer from (i) the sensitivity of the partition functions used to partition reachable states, and (ii) the high communication cost due to cross over transitions whose source and destination states are placed in different machines. Furthermore, most of these approaches focus on checking a program model with respect to some properties, whereas in the case of addition, we may need to revise a program model. As such, cross over transitions may cause even more complications.

Our proposed approach in this paper is based on a new paradigm, where we partition the code of a program into its actions instead of partitioning its reachability graph. Figure 1 depicts the basic idea behind the proposed approach. Each action is deployed on a separate machine in a parallel/distributed platform. Our approach comprises two layers, namely *ranking* and *addition* layers. In the ranking layer, we associate

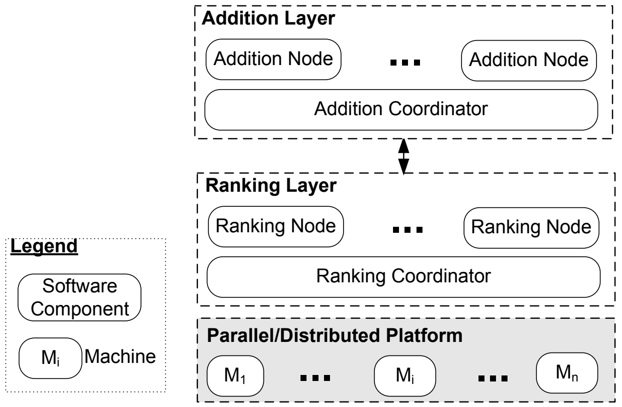

Figure 1: A large-scale framework for the addition of Leads-To properties.

a *Ranking Node* with each program action. Then, without creating an integrated model of the program, we analyze the contribution of each action in all the possible computation prefixes that reach Q. Ranking nodes are coordinated by a *ranking coordinator* in a distributed/parallel fashion towards identifying the set of states in each rank. The results of ranking will be used by the addition layer. We consider a similar architecture for the addition layer except that in the addition layer, we may revise individual actions using a *Addition Node* associated with each action. We note that the addition layer may invoke the ranking layer several times because the program model may be revised and the ranking should be repeated using the updated program model until the addition algorithm terminates. In the next section, we present the details of the algorithms behind the ranking and addition layers.

5 Action-Level Addition

In this section, we discuss the underlying algorithms of our action-level method. In Subsection 5.1, we present our action-level ranking algorithm. Then in Subsection 5.2, we illustrate how the results of the ranking layer are used by the addition layer. We present our approach in the context of the Token Passing example introduced in Section 2. (For ease of presentation, we use the sans serif font for the names of state predicates and functions.)

5.1 Ranking Layer

In this section, we present an algorithm that ranks each invariant state s of a program p based on the shortest computation prefix of p from s to a given destination predicate, denoted DestPred. Let $p = \langle V_p, \Pi_p \rangle$ be a program with an invariant Inv. The program p has N actions A_j, where $1 \leq j \leq N$, and N is a fixed positive integer. Our algorithm includes two components: Ranking_Node (see Figure 2) and Ranking_Coordinator (see Figure 3). A Ranking_Node component, denoted $rNode_j$, is instantiated for each program action $A_j : grd \rightarrow stmt$. The Ranking_Coordinator synchronizes the activities of all $rNode_j$. In each $rNode_j$, we first compute the intersection between DestPred and the program invariant Inv. This intersection includes all states with rank zero (see Lines 1-3 in Figures 2 and 3). Such states are placed in the first entry of a hash table of state predicates, denoted $RHash$. We also consider a state predicate explored that captures the set of states that have so far been ranked.

In each iteration of the while loop in Ranking_Node, each $rNode_j$ computes a set of states from where the DestPred calculated in the previous iteration can be reached in a single-step execution of the action A_j ($1 \leq j \leq N$). (In the start of the first iteration (i.e., $i = 1$), DestPred is equal to the predicate Q in $P \leadsto Q$.) Specifically, in Step 7 (see Figure 2), Ranking_Node calculates the set of transitions (s_0, s_1), where s_0 is an invariant state in which grd holds, s_0 is not in the set of already ranked states, and s_1 reaches DestPred by the execution of $stmt$. Moreover, the transition (s_0, s_1) should meet the write restrictions of A_j, denoted $wRest_j$. The function extractUnPrimed returns a state predicate R_{ij} representing the set of source states of

```
Ranking_Node($RHash$: hashtable; $grd \rightarrow stmt$: action; $wRest_j$: transition predicate;
                                        $Inv, DestPred$: state predicate)
// $j$ is the index of the $j$-th program action $A_j : grd \rightarrow stmt$
{
   - $DestPred := DestPred \wedge Inv$;                                          (1)
   - $i := 0$; /* Iterator */                                                    (2)
   - $RHash[i] := DestPred$;                                                     (3)
   - $explored := DestPred$;                                                     (4)
   while ($DestPred \neq \emptyset$) {                                           (5)
         - $i := i + 1$;                                                         (6)
         - $R_{ij}$ := extractUnPrimed($grd \wedge Inv \wedge \neg explored \wedge wRest_j \wedge stmtExpr \wedge$
                                                    $Primed(DestPred)$);         (7)
         - Send $R_{ij}$ to the coordinator;                                     (8)
         - Receive the new $DestPred$ from the coordinator;                      (9)
         - $RHash[i] := DestPred$;                                               (10)
         - $explored := explored \vee DestPred$                                  (11)
   }
}
```

Figure 2: The algorithm of the Ranking Node.

all transitions (s_0, s_1) computed in Step 7 in Figure 2. That is, R_{ij} captures the set of states from where states with rank $i-1$ can be reached with a single-step execution of A_j. Then $rNode_j$ sends its R_{ij} to Ranking_Coordinator. The Ranking_Coordinator calculates the disjunction of all R_{ij} predicates (see Lines 6-7 in Figure 3). The state predicate $\vee_{j=1}^{N} R_{ij}$ captures the set of states from where states with rank $i-1$ can be reached by any program action in a single step. Such states are added to the set of explored states. In fact, at the end of the i-th iteration, the state predicate explored captures all states whose rank is between 0 and i. Continuing thus, until all R_{ij} become empty in some iteration i, each $rNode_j$ would have a hash table $RHash$ such that $RHash[i]$ contains a state predicate representing all state with rank i.

```
Ranking_Coordinator($RHash$: hashtable; Inv, DestPred: state predicate)
/* $N$ is the number of program actions */
{
   - $DestPred := Inv \wedge DestPred$;                              (1)
   - $i := 0$; /* Iterator variable */                               (2)
   - $RHash[i] := DestPred$;                                         (3)
   - while ($DestPred \neq \emptyset$) {                             (4)
         - $i := i + 1$;                                             (5)
         - Forall $1 \leq j \leq N$, receive $R_{ij}$ from $rNode_j$;   (6)
         - $DestPred := \vee_{j=1}^{N} R_{ij}$                       (7)
         - Forall $1 \leq j \leq N$, send $DestPred$ to $rNode_j$;   (8)
         - $RHash$[i] := $DestPred$;                                 (9)
   }
}
```

Figure 3: The algorithm of the Ranking Coordinator.

Theorem 5.1 Action-level ranking terminates.
Proof. In order to prove the termination of the ranking phase, we first make the following observation that for each $rNode_j$ and in each iteration i of the while loop in Figure 2, the intersection of the state predicate R_{ij} and the state predicate explored is empty. Since the Ranking_Coordinator computes the union of all R_{ij} in each iteration i, it follows that, before Step 11 in Figure 2, we have $DestPred \cap explored = \emptyset$. That is, in each iteration, the size of the explored predicate increases in Step 11 in Figure 2. Since the state space of a program p (i.e., $\mathcal{Q}_p$) is finite, the explored predicate can at most become equal to $\mathcal{Q}_p$. Therefore, based on the observation that $DestPred \cap explored = \emptyset$, it follows that $DestPred = \emptyset$ will eventually hold. □

Theorem 5.2 Action-level ranking is sound; i.e., $RHash[i]$ contains the set of states from where the shortest

computation prefix to DestPred has length i.
Proof. We consider the following loop invariant for the while loop in Figure 2:

Loop Invariant: $RHash[i]$ denotes the set of states from where the shortest computation prefix of program p has length i .

- *Initialization.* Before the while loop in Figure 2 starts, we have $RHash[0] = DestPred$, which preserves the loop invariant.

- *Maintenance.* Now, we show that if the loop invariant holds before the i-th iteration of the while loop, then it also holds before the $(i+1)$-th iteration. Step 6 in Figure 2 only increases the value of the loop counter i and does not affect the contents of $RHash[i]$. Step 7 computes the set of all states $R_{(i+1)j}$ from where an action A_j can take the state of the program to the set of states explored so far in a single step. By construction, the states in $R_{(i+1)j}$ should reach a state in $\vee_{j=1}^{N} R_{ij}$ by a single transition; otherwise they must have been included in some R_{kj}, for $k \leq i$. Since DestPred is the union of $R_{(i+1)j}$ for $1 \leq j \leq N$, it follows that DestPred in iteration $i+1$ contains the set of states whose shortest computation prefix to the initial DestPred has length $i+1$. Therefore, before the next iteration, the loop invariant holds.

- *Termination.* When the while loop terminates, each $RHash[i]$, for $1 \leq i \leq RHash.size$, has the set of states whose rank is equal to i. □

TP Example. Consider the property $\mathcal{L}_1$ that requires the reachability to states where only one process has a token; i.e., $true \leadsto \mathcal{S}_1$. In this case, DestPred is equal to $\mathcal{S}_1$. Hence, we rank all states based on the shortest computation prefix of the TP program that reaches a state in $\mathcal{S}_1$. Towards this end, we create three ranking nodes corresponding to each action A_j, denoted $rNode_j$ $(1 \leq j \leq 3)$. In the first iteration of the while loop, $rNode_1$ computes the set of states (denoted R_{11}) from where the execution of the action A_1 would take the state of the program from a state with two/three tokens to a state where only one token exists. The state predicate R_{11} is equal to $(x_1 \neq x_2) \wedge (x_2 = (x_3 \oplus 1))$. To illustrate this, consider a transition (s_0, s_1) captured by A_1 where there are two or three tokens in s_0 and only one process has a token in s_1. We observe that the value of x_1 and x_3 cannot be the same in s_1. Thus, P_1 does not have a token in s_1. By contradiction, if $(x_2 \neq (x_3 \oplus 1))$ in s_0, then $x_1 \neq x_2$ holds in s_1 because A_1 assigns $(x_3 \oplus 1)$ to x_1. Thus, P_2 has a token in s_1. As a result, we must have had $x_2 = x_3$ in s_0 since only one token can exist in s_1. If $(x_2 = x_3)$ holds in s_0 and we start either with $x_1 = x_2$ or with $x_1 \neq x_2$ in s_0, then only one token can exist in s_0, which is a contradiction with our assumption that two or three tokens exist in s_0. Hence, $(x_2 = (x_3 \oplus 1))$ must hold in s_0. As a result, x_1 will become equal to x_2 after the execution of A_1. We note that $x_1 = x_2$ cannot be true in s_0 because having both $x_1 = x_2$ and $(x_2 = (x_3 \oplus 1))$ in s_0 means that only P_3 has a token in s_0, which again contradicts with having multiple tokens in s_0. Therefore, $(x_1 \neq x_2) \wedge (x_2 = (x_3 \oplus 1))$ must hold in s_0. For example, consider the state $\langle 1, 2, 1 \rangle$, where (1) P_1 has a token because $x_1 = x_3$; (ii) P_2 has a token because $x_1 \neq x_2$, and (3) P_3 has a token because $x_2 \neq x_3$. The execution of A_1 from $\langle 1, 2, 1 \rangle$ would change the state of the TP program to $\langle 2, 2, 1 \rangle$, where only P_3 has a token.

The state predicate $R_{12} = R_{13} = ((x_1 \neq x_2) \wedge (x_2 \neq x_3))$ represents the states from where the execution of A_2 or A_3 would get the TP program from a state with multiple tokens to a state where only one process has a token. Thus, in the first iteration, $rNode_2$ and $rNode_3$ would respectively send the state predicates R_{12} and R_{13} to the ranking coordinator. The predicates R_{21}, R_{22} and R_{23} evaluate to the empty set, thereby terminating the ranking process in the second iteration.

5.2 Addition Layer

In this section, we illustrate how we use the results of the ranking layer to add Leads-To properties to parallel programs. Similar to the ranking layer, in the addition layer, we create an instance of the Addition_Node component (see Figure 4) corresponding to each program action A_j, denoted $aNode_j$. Each $aNode_j$ collaborates with an instance of the Addition_Coordinator (see Figure 5) during addition.

The addition starts with invoking the ranking layer to identify the rank of each state inside the program invariant Inv with respect to Q as the destination predicate (Line 1 in Figures 4 and 5). Then each $aNode_j$ enters a loop in which it eliminates two classes of transitions from the action A_j (Recall from Section 2 that each action in fact represents a set of transitions.): (1) transitions that reach a state s from where Q cannot be reached by program computations; i.e., $Rank(s) = \infty$ (see Steps 3 and 4 in Figure 4), and (2) transitions that reach a deadlock state created due to removing some transition of the first class (see Step 8 in Figure 4). If a program computation reaches a deadlock state or a state with rank infinity, then reachability of

```
Addition_Node( grd → stmt: action; Inv, P, Q: state predicate; $wRest_j$: transition predicate)
{
/* $j$ is the index of the $j$-th program action $A_j : grd \rightarrow stmt$ */
/* P and Q are state predicates in the leads-to property $\mathsf{P} \leadsto \mathsf{Q}$ */
/* $RHash[i]$ denotes states from where the shortest computation prefix to $Q$ has length $i$. */

  - Rank_Node($RHash$, $grd \rightarrow stmt$, $Inv, Q$);                                   (1)
  - $m := RHash.size - 1$;                                                                   (2)
  - $InfinityRank := (P \wedge Inv) - (\vee_{k=0}^{m} RHash[k])$;                            (3)

  repeat {
      - Exclude($grd \wedge Inv \wedge wRest_j \wedge stmtExpr \wedge Primed(InfinityRank), grd \rightarrow stmt$);   (4)
      - Send $grd_j$ to the Addition Coordinator;                                            (5)
      - Receive a new invariant $Inv_{new}$ and $Deadlock$ states;                           (6)
      - $Inv := Inv_{new}$;                                                                  (7)
      - Exclude($grd \wedge Inv \wedge wRest_j \wedge stmtExpr \wedge Primed(Deadlock), grd \rightarrow stmt$);   (8)
      - Rank_Node($RHash, grd \rightarrow stmt, Inv, Q$);                                    (9)
      - $m := RHash.size - 1$;                                                               (10)
      - $InfinityRank := (P \wedge Inv) - (\vee_{k=0}^{m} RHash[k])$;                        (11)
  } until ((($P \wedge Inv \wedge InfinityRank$) is unsatisfiable) $\vee$ ($grd$ is unsatisfiable));   (12)

  - if ($grd$ is unsatisfiable) then declare that this action should be removed; exit;      (13)
  - For ($i := 1$ to $m - 1$)
      - Exclude($grd \wedge Inv \wedge wRest_j \wedge RHash[i] \wedge$
                        $stmtExpr \wedge Primed(\vee_{k=i+1}^{m} RHash[k]), grd \rightarrow stmt$);   (14)
  - For ($i := 1$ to $m$)
      - Exclude($grd \wedge Inv \wedge wRest_j \wedge RHash[i] \wedge$
                        $stmtExpr \wedge Primed(RHash[i]), grd \rightarrow stmt$);            (15)
}

Exclude($transPred$: transition predicate, $A_j : grd \rightarrow stmt$: action)
// exclude the set of transitions represented by $transPred$ from the action $grd \rightarrow stmt$.
{     - $transPred := grd \wedge stmtExpr \wedge \neg transPred \wedge wRest_j$;
      - $grd :=$ extractUnPrimed($transPred$);                                               }
```

Figure 4: The algorithm of the Addition Node.

Q cannot be guaranteed. Then, each $aNode_j$ sends its updated guard to the coordinator. The addition coordinator receives the updated guards of all nodes (Step 2 in Figure 5) and calculates the set of invariant states where no updated guard is enabled. Such states have no outgoing transitions; i.e., deadlocked states (Step 3 in Figure 5). Since some invariant states may become deadlocked (due to Step 4 in Figure 4), the addition coordinator computes a new invariant that excludes deadlock states. The coordinator sends the new invariant and the set of deadlock states to all addition nodes. Afterwards, each addition node eliminates transitions that reach a deadlock state ensuring the deadlock-freedom requirement. Since at this point the program actions might have been updated, the ranking should be repeated. Each $aNode_j$ performs the above steps until either all states in P have a finite rank, or the action associated with $aNode_j$ is eliminated (i.e., the guard of action A_j becomes empty).

After ensuring that no program computation starting in a state in P would reach a deadlock state or a infinity-ranked state, we ensure that all non-progress cycles would be resolved. We classify such non-progress cycles into two groups: (1) cycles whose states span over distinct ranks, and (2) cycles whose all states belong to the same rank. The For loop in Step 14 of Figure 4 resolves the first group of cycles by removing any transition (s_0, s_1), where $Rank(s_0) < Rank(s_1)$. Respectively, the For loop in Step 15 of Figure 4 resolves the second group of cycles by removing any transition (s_0, s_1), where s_0 and s_1 are in the same rank. Note that Steps 14 and 15 do not create additional deadlock states because a computation prefix that reaches Q

```
Addition_Coordinator(Inv, P, Q: state predicate) {
/* $aNode_j$ denotes the addition node for the $j$-th action. */
/* $N$ is the number of program actions. */
/* $RHash$ is a hashtable. */

 - Ranking_Coordinator($RHash, Inv, Q$);                                    (1)
 repeat {
     - Forall $1 \leq j \leq N$, receive guards $grd_j$ from $aNode_j$;           (2)
     - $Deadlock := Inv - (\vee_{j=1}^{N} grd_j)$;                            (3)
     - $Inv_{new} := Inv - Deadlock$;                                       (4)
     - Forall $1 \leq j \leq N$, send $Inv_{new}$ and $Deadlock$ to $aNode_j$;    (5)
     - Ranking_Coordinator($RHash, Inv, Q$);                                (6)
     - $Inv := Inv_{new}$;                                                  (7)
 } until ((($P \wedge Inv$) is unsatisfiable) $\vee$ ($Deadlock$ is unsatisfiable));

 - If ($Inv$ is unsatisfiable) then declare that addition is not possible; exit;   (8)
 - declare that addition succeeded;                                          (9)
}
```

Figure 5: The algorithm of the Addition Coordinator.

is originated at s_0; i.e., s_0 has an outgoing transition.

Theorem 5.3 Action-level addition is sound.
Proof. We show that the program synthesized by all $aNode_j$, for $1 \leq j \leq N$, meets the requirements of the problem of adding Leads-To properties. Let p^r denote the synthesized program and $\mathcal{I}^r$ denote its new invariant.

- Since Addition_Coordinator updates the invariant of the input program, denoted $Inv = \mathcal{I}$, by excluding some deadlock states from Inv (see Step 4 in Figure 5), it follows that the returned invariant is a subset of the input invariant; i.e., $\mathcal{I}^r \Rightarrow Inv$.

- Each $aNode_j$ ($1 \leq j \leq N$) updates its associated action A_j in Steps 4, 8, 14 and 15 of Figure 4 by excluding some transitions from the set of transitions captured by A_j; i.e., no new transitions are added to A_j. Therefore, the returned action of $aNode_j$ meets the second requirement of the addition problem.

- Since all deadlock states are eliminated from Inv in Step 4 in Figure 5, the Addition_Coordinator algorithm returns an invariant that does not include any deadlocks. Moreover, based on the above steps of the proof, $\mathcal{I}^r$ does not include any new states, and the actions of p^r do not include new transitions. It follows that the computations of p^r from $\mathcal{I}^r$ are a subset of the computations of the input program starting in $\mathcal{I}$. Thus, p^r satisfies $Spec$ from $\mathcal{I}^r$. Otherwise, there must exist a computation of p that violates $Spec$ from $\mathcal{I}^r \Rightarrow \mathcal{I}$, which is a contradiction with p satisfying $Spec$ from $\mathcal{I}$.

  Now, if p^r violates the Leads-To property $P \rightsquigarrow Q$ from $\mathcal{I}^r$, then, by the definition of Leads-To, there must be a computation $\sigma = \langle s_0, s_1, \cdots \rangle$ of p^r that starts in $P \wedge \mathcal{I}^r$ and never reaches a state in Q. There are two possible scenarios for σ: either σ deadlocks in a state $s \notin Q$ before reaching Q, or σ reaches a non-progress cycle. The former cannot be the case because, based on the above argument, all computations of p^r are infinite. Consider σ to be the sequence $\langle s_0, s_1 \cdots, s_j, \cdots, s_k, s_j \cdots \rangle$, where $s_0 \in P$ and no state in the subsequence $\delta = \langle s_1 \cdots, s_j, \cdots, s_k, s_j \rangle$, $k \leq j$, belongs to Q.[4] If there exists a state s_m, where $1 \leq m \leq k$, in δ whose rank is infinity, then there is a program action A_l ($1 \leq l \leq N$) that induces the transition (s_{m-1}, s_m); i.e., the output action of $aNode_l$ includes a transition that reaches a infinity-ranked state. By construction, all such transitions are removed by Addition_Node. Thus, all states in δ have a finite rank.

  If $Rank(s_k) = Rank(s_j)$, then there must be some program action that executes the transition (s_k, s_j), which is a contradiction with Step 15 in Figure 4. If $Rank(s_k) < Rank(s_j)$, then some action of p^r

[4]We assume P does not intersect Q.

includes a transition from a lower-ranked state to a higher-ranked state, which is a contradiction with Step 14 in Figure 4. If $Rank(s_k) > Rank(s_j)$, then since s_k is reachable from s_j, there must be some transition (s_m, s_{m+1}), for $j \leq m < k$, where $Rank(s_m) < Rank(s_{m+1})$, which is again a contradiction with Step 14 in Figure 4. Therefore, the assumption that σ violates $P \leadsto Q$ is invalid. $\Box$

Theorem 5.4 Action-level addition is complete.
Proof. Let p be a program with an invariant $\mathcal{I}$ such that p satisfies its specification $Spec$ from $\mathcal{I}$. Moreover, let $P \leadsto Q$ be the Leads-To property that should be added to p. By contradiction, consider the case where the action-level addition fails to add $P \leadsto Q$, and there exists a program p' with an invariant $\mathcal{I}'$ that meets the requirements of the addition problem (in Section 3). Thus, $\mathcal{I}'$ is non-empty, and we have $\mathcal{I}' \Rightarrow \mathcal{I}$. Moreover, p' must satisfy $(Spec \wedge (P \leadsto Q))$ from $\mathcal{I}'$. That is, all computations of p' are infinite and satisfy $(Spec \wedge (P \leadsto Q))$ from $\mathcal{I}'$. Hence, there exists a subset of $\mathcal{I}$ and a set of revised program actions that meet the requirements of the addition problem. Our algorithm declares failure only if no such invariant and set of revised actions exist. Therefore, our algorithm would have found p' and $\mathcal{I}'$. $\Box$

TP Example. The first invocation of the ranking layer in Step 1 of each $aNode$ identifies 28 states in rank 0 (captured by $DestPred \equiv \mathcal{S}_1$) and 36 states in rank 1, which comprise the total state space of the TP program. Thus, there are two ranks (i.e., $m = 1$), and no states with rank infinity exist. (This is an intuitive result as actions A_1-A_3 have no guards; i.e., from every state in the state space, there exists a computation prefix to $\mathcal{S}_1$.) Since we have $InfinityRank = \emptyset$, no transitions are excluded from the actions in Step 4 of each $aNode_j$; i.e., no $aNode_j$ updates its guard grd. Thus, the invariant of TP remains unchanged and the set of deadlock states is empty in the first iteration of the repeat-until loop in Figure 4. Since in the first iteration no program action is changed, the second call to the ranking layer would give us the same result as the first invocation. The loop terminates after the first iteration as $InfinityRank = \emptyset$.

The first For loop in Step 14 does not eliminate any transitions because there are no ranks higher than rank 1. Note that we do not remove the transitions that start in rank 0 and end in higher ranks since they originate at DestPred; i.e., such transitions cannot be in a non-progress cycle. For example, the action A_1 includes transitions that start in $\mathcal{S}_1$, where only one token exists, and terminate in states where multiple tokens exist. Consider a transition from $s_0 = \langle 2, 2, 0 \rangle$ to $s_1 = \langle 1, 2, 0 \rangle$ executed by action A_1, where only P_3 has a token in s_0, but both P_2 and P_3 have tokens in s_1. Nonetheless, the execution of actions A_2 and A_3 from a state where there is only one token would not generate additional tokens.

The For loop in Step 15 eliminates the transitions that start and end in rank 1. For example, both states $s_2 = \langle 0, 2, 0 \rangle$ and $s_3 = \langle 1, 2, 0 \rangle$ belong to rank 1 (because more than one tokens exist in s_2 and s_3) and the action A_1 executes the transition (s_2, s_3). Moreover, consider a self-loop from $\langle 3, 0, 2 \rangle$ to itself executed by A_1. This self-loop in rank 1 is indeed a non-progress cycle. The following parameterized action represents the revised actions A'_2 and A'_3 in the program TP' synthesized by PARDES ($i = 2, 3$):

$$A'_i : \quad x_i \neq x_{i-1} \quad \longrightarrow \quad x_i := x_{i-1};$$

The following action is the revised version of action A_1:

$$A'_1 : \; x_3 \neq (x_1 \oplus 3) \;\longrightarrow\; x_1 := (x_3 \oplus 1);$$

While TP' satisfies $true \leadsto \mathcal{S}_1$, it does not guarantee that a stable state, where all x values are equal, will be reached. Formally, TP' does not satisfy $\mathcal{S}_1 \leadsto \mathcal{S}_2$. In a non-progress scenario, process P_1 changes the state of TP' from $\langle 1, 1, 2 \rangle$ to $\langle 3, 1, 2 \rangle$ from where a non-progress cycle starts with an execution order P_3, P_2, P_1 repeated for 8 rounds; none of the states in this cycle is a stable state. (We leave it to the reader to validate this claim.) We have used PARDES to add $\mathcal{S}_1 \leadsto \mathcal{S}_2$ to TP' while preserving $true \leadsto \mathcal{S}_1$. The actions of the resulting program TP" are as follows:

$$\begin{array}{lll}
A''_1 : & (x_3 \neq (x_1 \oplus 3)) \wedge (x_2 \neq (x_1 \oplus 1)) & \\
 & & \longrightarrow \; x_1 := (x_3 \oplus 1); \\
A''_2 : & (x_1 \neq x_2) & \longrightarrow \; x_2 := x_1; \\
A''_3 : & (x_3 \neq x_2) \wedge (x_1 = x_2) & \longrightarrow \; x_3 := x_2;
\end{array}$$

6 Examples

In this section, we present two additional case studies of the application of our action-level approach in automated addition of Leads-To properties to parallel programs. In Section 6.1, we illustrate how we have synthesized a Barrier Synchronization program. In Section 6.2, we add progress to a mutual exclusion program.

6.1 Barrier Synchronization

In this section, we demonstrate our approach in the design of a barrier synchronization program (adapted from [28]). The Barrier program includes three processes P_1, P_2, and P_3. Each process P_j, $1 \leq j \leq 3$, has a variable pc_j that represents the program counter of P_j. The program counter pc_j can be in three positions *ready, execute, success.* The actions of process P_j in the initial program are as follows ($j = 1, 2, 3$):

$$\begin{array}{llcl}
A_{j1}: & pc_j = ready & \longrightarrow & pc_j := execute; \\
A_{j2}: & pc_j = execute & \longrightarrow & pc_j := success; \\
A_{j3}: & pc_j = success & \longrightarrow & pc_j := ready;
\end{array}$$

If process P_j is in the ready position, then it changes its position to execute. From execute, the position of P_j transitions to success, and finally back to the ready position from success. Let $\langle pc_1, pc_2, pc_3 \rangle$ denote a state of the Barrier program. The problem statement requires that, starting from the state $allR = \langle ready, ready, ready \rangle$, all processes will eventually synchronize in the state $allS = \langle success, success, success \rangle$. The invariant of the Barrier program, denoted $\mathcal{I}_B$, specifies that at least two processes should be in the same position, where

$$\mathcal{I}_B = (\forall j : 1 \leq j \leq 3 : (pc_j \neq ready)) \vee (\forall j : 1 \leq j \leq 3 : (pc_j \neq execute)) \vee (\forall j : 1 \leq j \leq 3 : (pc_j \neq success))$$

In the Barrier program, one process could circularly transition between the positions ready, execute, success, thereby preventing the synchronization on $\langle success, success, success \rangle$. To ensure synchronization, we add the property $allR \leadsto allS$ to the above program. The actions of P_1 in the synthesized program are as follows:

$$\begin{array}{lllcl}
A'_{11}: & (pc_1 = ready) \wedge (((pc_2 \neq success) \wedge (pc_3 \neq success)) \vee ((pc_2 = success) \wedge (pc_3 = success))) & & & \\
& & & \longrightarrow & pc_1 := execute; \\
A'_{12}: & (pc_1 = execute) \wedge (pc2 \neq ready) \wedge (pc3 \neq ready) & & \longrightarrow & pc_1 := success; \\
A'_{13}: & (pc_1 = success) \wedge (((pc_2 = ready) \wedge (pc_3 = ready)) \vee ((pc_2 = success) \wedge (pc_3 = success))) & & & \\
& & & \longrightarrow & pc_1 := ready;
\end{array}$$

The synthesized actions of P_2 and P_3 are structurally similar to those of P_1. To gain more confidence in the implementation of PARDES, we have model checked the barrier synchronization and the token passing programs using the SPIN model checker [29]. (The Promela code is available at `http://www.cs.mtu.edu/~aebnenas/files/{TP.txt,Barrier.txt}`).

6.2 Mutual Exclusion: Adding Progress

In this section, we use our approach to automatically synthesize Peterson's solution [30] for the mutual exclusion problem for two competing processes. Specifically, we start with a program that meets its safety requirement (i.e., the two process will never have simultaneous access to shared resources), but does not guarantee progress for competing processes (i.e., if a process tries to obtain the shared resources, then it will eventually gain control over shared resources). The Mutual Exclusion (ME) program has two processes denoted P_0 and P_1. Each process P_j ($j = 0, 1$) can be in three synchronization states, namely *critical* state, where P_j uses some resources that are shared with the other process, *trying* state, where P_j tries to enter its critical state, and *non-trying* state, where P_j does not intend to enter its critical state. Process P_j has three Boolean variables cs_j, ts_j, and ns_j that respectively represent its critical, trying and non-trying states.

Moreover, P_j has a Boolean variable $flag_j$ representing its intention for using shared resources. Processes take turn to use shared resources, which is represented by an integer variable $turn$ with the domain $\{0, 1\}$. The actions of P_j, for $j = 0, 1$, in the input program are as follows. We note that, in this section, $\oplus$ denotes addition in modulo 2.

$$\begin{array}{lll}
A_{j1}: & ns_j & \longrightarrow & ts_j := true; \\
 & & & flag_j := true; \\
 & & & turn := j \oplus 1; \\
 & & & ns_j := flase; \\
A_{j2}: & ts_j \wedge \neg flag_{j\oplus 1} & \longrightarrow & cs_j := true; \\
 & & & ts_j := false; \\
A_{j3}: & cs_j & \longrightarrow & flag_j := false; \\
 & & & cs_j := false; \\
 & & & ns_j := true;
\end{array}$$

If a process P_j is in its non-trying state, then it enters its trying state and illustrates its intention by setting $flag_j$ to true. However, P_j gives the priority to the other process by setting $turn$ to $(j \oplus 1)$. In its trying state, if the other process does not intend to use the shared resources, then P_j transitions to its critical state. The process P_j goes to its non-trying state from its critical state. The ME program guarantees that the two competing processes will not be in their critical state simultaneously. Formally, the invariant of the ME program is equal to the state predicate $\neg(cs_0 \wedge cs_1)$. However, inside the invariant, a process may repeat the above steps in a cycle and deprive the other process from entering its critical state. In other words, the above program does not satisfy $ts_j \leadsto cs_j$, for $j = 0, 1$; i.e., a process that is trying to enter its critical state may never get a chance to transition to its critical state and benefit from shared resources. After adding the above Leads-To property to the ME program, we have synthesized a new program, denoted ME'. The actions of process P_j in ME' are as follows:

$$\begin{array}{llll}
A'_{j1}: & ns_j & \longrightarrow & ts_j := true; \\
 & & & flag_j := true; \\
 & & & turn := j \oplus 1; \\
 & & & ns_j := flase; \\
A'_{j2}: & ts_j \wedge (\neg flag_{j\oplus 1} \vee (\mathbf{turn} = \mathbf{j})) & \longrightarrow & cs_j := true; \\
 & & & ts_j := false; \\
A'_{j3}: & cs_j & \longrightarrow & flag_j := false; \\
 & & & cs_j := false; \\
 & & & ns_j := true;
\end{array}$$

The program ME' is the same as Peterson's solution [30] for two competing processes. We are currently investigating the application of our approach in the large-scale design of Dijkstra's solution [31] for $n > 2$ competing processes.

7 Experimental Results

While we have added Leads-To properties to several programs, in this section, we focus on the results of adding Leads-To to the barrier synchronization program. Specifically, we illustrate how we added $\mathcal{L}_B \equiv allR \leadsto allS$ to a barrier synchronization program with 18 processes on a cluster of 5 regular PCs connected to the Internet.

Platform of experiments. The underlying hardware platform of our experiments comprises five regular PCs with the following specifications.

Machine M_1 is connected to the Internet via a fast Ethernet subnet, and M_2-M_5 are connected to the Internet via a wireless network. While the ranking/addition nodes could have different implementations, we have implemented the current version of PARDES in C++ and we have used BDDs [9] to model programs and state/transition predicates.

Experiments. We added $\mathcal{L}_B$ to several instances of the Barrier program introduced in Section 6.1 with different number of processes from 3 to 18. First, we designed the program in Section 6.1, where all ranking

Machines	CPU	RAM	Operating System
M_1	Dual Core, 3.2 GHz	2 GB	Linux (Fedora)
M_2	Pentium M, 1.86 GHz	502 MB	Win XP 2002
M_3	Pentium M, 1.86 GHz	502 MB	Win XP 2002
M_4	Dual Core, 1.83 GHz	1 GB	Win XP 2002
M_5	Pentium M, 2 GHz	1 GB	Win XP 2002

Figure 6: Machines used in the experiments.

and addition nodes were run as independent processes on M_1. Afterwards, we synthesized Barrier programs with 6, 9 and 15 processes on M_1. In order to investigate the cost of synthesis, communication and ranking in PARDES, we repeated the addition of $\mathcal{L}_B$ to the 15-process Barrier once on M_1 and M_2, and once on M_1, M_2 and M_3. Figure 7 summarizes the results of these experiments.

Machines	Synthesis Time	Communication Time	Ranking Time
M_1	198 Sec.	203 Sec.	785 Sec.
M_1, M_2	412 Sec.	415 Sec.	1639 Sec.
M_1, M_2, M_3	572 Sec.	575 Sec.	2281 Sec.

Figure 7: Results of adding Leads-To to the Barrier program with 15 processes.

Notice that, in this case, action-level addition on a cluster of machines degrades the time efficiency of addition. Moreover, the cost of ranking overtakes the cost of synthesis. One reason behind the high cost of ranking is the current implementation of Ranking_Coordinator as a first-in/first-out server. We are investigating an implementation of Ranking_Coordinator as a concurrent server, which would potentially improve the time efficiency of PARDES. While the ranking overhead undermines the overall time efficiency of addition, we note that our first priority is the ability to easily scale up PARDES once the addition fails due to space complexity.

The addition of $\mathcal{L}_B$ to Barrier with 18 processes failed on M_1 due to space complexity. As a result, we scaled up the addition to M_1, M_2 and M_3, but this experiment was also unsuccessful for the same reason. When we included all 5 machines in the experiment, we were able to automatically add $\mathcal{L}_B$ to the 18-process Barrier in about 115 minutes. The results of this section illustrate that automatic design of the Barrier program with 18 processes would not have been possible without the help of PARDES in scaling the addition up to a cluster of 5 machines.

8 Related Work

In this section, we discuss related work on techniques for reducing the space complexity of automated design/verification. Several approaches exist for compositional design of concurrent programs [12, 13, 32, 33] and for parallel and distributed model checking [34]. Smith [32] decomposes the program specifications into the specifications of sub-problems and combines the results of synthesizing solutions for sub-problems. Puri and Gu [33] propose a divide-and-conquer synthesis method for asynchronous digital circuits. Abadi and Lamport [12] present a theory for compositional reasoning about the behaviors of concurrent programs. Giannakopoulou *et al.* [13] present an approach for compositional verification of software components. Edelkamp and Jabbar [34] divide the model checking problem so it can be solved on a parallel platform.

Previous work on automatic addition of fault-tolerance [7, 23, 35–37] mostly focuses on techniques for reducing time/space complexity of synthesis. For example, Kulkarni *et al.* [35] present a set of heuristics based on which they reduce the time complexity of adding fault-tolerance to integrated models of distributed programs. Kulkarni and Ebnenasir [7] present a technique for reusing the computations of an existing fault-tolerant program in order to enhance its level of tolerance. They also present a set of pre-synthesized fault-tolerance components [36] that can be reused during the addition of fault-tolerance to different programs. We have presented a SAT-based technique [23] where we employ SAT solvers to solve some verification problems during the addition of fault-tolerance to *integrated* program models. Bonakdarpour and Kulkarni [37] present a symbolic implementation of the heuristics in [35] where they use BDDs to model distributed programs.

Our proposed method differs from aforementioned approaches in that we focus on the problem of adding PLTL properties to programs instead of verifying PLTL properties. The action-level addition proposed in this paper adopts the decomposition approach that we present in [38] for the addition of fault tolerance properties. Nonetheless, in this paper, our focus is on automated addition of Leads-To properties instead of adding fault tolerance. While the asymptotic time complexity of adding a single Leads-To property in PLTL is not worse than verifying that property [1], in practice, the space complexity of addition seems to be higher than verification. One reason behind this is that the program model at hand may be revised, which makes it very difficult to devise efficient on-the-fly addition algorithms that are complete. We note that while our focus is on action-level *addition* of liveness properties, our approach can potentially be used for action-level *verification* of liveness properties.

9 Conclusions and Future Work

We presented an action-level method for adding Leads-To properties to shared memory parallel programs. In our approach, we start with a parallel program that meets its specification, but may not satisfy a new Leads-To property raised due to new requirements. Afterwards, we partition the code of the program into its set of actions, and analyze (and possibly revise) each action separately in such a way that the revised program satisfies the new Leads-To property and preserves its specification. We have also developed a distributed framework, called PARallel program DESigner (PARDES), that exploits the computational resources of a network of workstations for automated addition of Leads-To to shared memory parallel programs. Our preliminary experimental results are promising as we have been able to easily scale up the addition of Leads-To to a barrier synchronization protocol on a cluster of 5 regular PCs.

We are currently investigating several extensions of our action-level addition approach. First, we are developing a family of action-level algorithms for the addition of PLTL properties to parallel programs. Such a family of algorithms provides the theoretical background of a large-scale framework that facilitates (and automates) the design of parallel programs for mainstream programmers. Second, we would like to integrate our approach in modeling environments such as UML [39] and SCR [40], where visual assistance is provided for developers while adding PLTL properties to programs. Third, we are studying the action-level addition of PLTL properties to distributed programs, where processes have read/write restrictions in a distributed shared memory model. Finally, we plan to investigate the application of our approach in distributed model checking of parallel/distributed programs.

References

[1] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.

[2] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[3] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. In *CAV*, pages 419–427, 1996.

[4] N. Wallmeier, P. Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *CIAA, LNCS, Vol. 2759*, pages 11–22, 2003.

[5] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125 – 185, 2004.

[6] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[7] Sandeep S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 441–449, 2003.

[8] Ali Ebnenasir, Sandeep S. Kulkarni, and Anish Arora. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal of Software Tools for Technology Transfer (To appear)*, 2008.

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[10] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In *ASE*, pages 3–12, 2000.

[11] V. Stolz and F. Huch. Runtime verification of concurrent haskell programs. *Electronic Notes in Theoretical Computer Science*, 113:201–216, 2005.

[12] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):73–132, 1993.

[13] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, 12(3), 2005.

[14] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic. Notes on Theoretical Computer Science*, 89(2), 2003.

[15] Arshad Jhumka and Neeraj Suri. Designing efficient fail-safe multitolerant systems. In *Formal Techniques for Networked and Distributed Systems*, pages 428–442, 2005.

[16] Sandeep S. Kulkarni and Ali Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, 2005.

[17] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[18] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, 2006.

[19] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 58(3):310–324, 2005.

[20] E. A Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics, pages 995–1067. Elsevier Science Publishers B. V., 1990.

[21] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[22] Ali Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.

[23] Ali Ebnenasir and Sandeep S. Kulkarni. SAT-based synthesis of fault-tolerance. *Fast Abstracts of the International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy, June 28 - July 1*, 2004.

[24] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[25] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[26] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In *CAV'97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278, 1997.

[27] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proceedings of International SPIN Workshops*, pages 22–39, 1999.

[28] S. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, October 1997.

[29] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[30] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[31] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[32] D.R. Smith. A problem reduction approach to program synthesis. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 32–36, 1983.

[33] R. Puri and J. Gu. A divide-and-conquer approach for asynchronous interface synthesis. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 118–125, 1994.

[34] Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking LTL. In *13th International SPIN Workshop on Model Checking of Software*, pages 1–18, 2006.

[35] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems*, pages 130 – 139, 2001.

[36] Sandeep S. Kulkarni and Ali Ebnenasir. Adding fault-tolerance using pre-synthesized components. *Fifth European Dependable Computing Conference (EDCC-5), LNCS, Vol. 3463, p. 72*, 2005.

[37] B. Bonakdarpour and Sandeep S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs. In *IEEE International Conference on Distributed Computing Systems*, pages 3–10, 2007.

[38] Ali Ebnenasir. DiConic addition of failsafe fault-tolerance. In *IEEE/ACM international conference on Automated Software Engineering*, pages 44–53, 2007.

[39] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[40] C. L. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS 1995)*, pages 109–122, 1995.