Automatic Addition of Liveness[1]

Ali Ebnenasir Sandeep S. Kulkarni

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

We present an incremental synthesis method where we improve the behaviors of finite-state concurrent programs by automatic addition of new liveness properties. Specifically, we present a sound and complete algorithm for adding Leads-to *properties to programs in the high atomicity model where processes can read/write all program variables in an atomic step. Since we automatically add new properties to programs, the synthesized program is correct by construction. Our incremental approach has the potential to (i) preserve the efficiency of the input program, and (ii) reuse the efforts put in the design of a program for the synthesis of its improved version. This issue is important in the maintenance of software systems where developers add new properties to a program while preserving its existing properties. As an illustration of our synthesis algorithm, we have synthesized mutual exclusion and readers-writers programs by incremental addition of* Leads-to *properties.*

Keywords: Liveness property, Automatic addition of liveness, Formal methods, Program synthesis

[1]Email: ebnenasi@cse.msu.edu, sandeep@cse.msu.edu
Web: `http://www.cse.msu.edu/~{ebnenasi,sandeep}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

1 Introduction

Automatic synthesis of programs that are correct by construction is desirable in a variety of disciplines (e.g., design of mission-critical systems, network protocols, operating systems). Such an automated synthesis guarantees that the synthesized program behaves as specified. Especially, in the design of reactive programs that have non-terminating computations the synthesis problem becomes even more challenging. In this paper, we present an incremental synthesis approach where we automatically add liveness properties to reactive programs while preserving their existing behaviors.

In the synthesis of reactive programs, previous research mostly concentrates on the synthesis of programs from their specification [1–8]. In such specification-based synthesis approaches, if one adds new properties to the specification then a new program should be re-synthesized from scratch using the new specification. The inability to reuse the effort put in the synthesis of a program for the synthesis of its improved version is an obstacle in the development of practical synthesis methods and tools. Especially, in the cases where the state space of the given program is large, it is necessary to reuse the computational structure of programs in the synthesis of their improved version.

The specification of reactive programs consists of safety and liveness properties [9]. There exist approaches in the literature for incremental addition of safety properties to reactive programs [10–15]. These approaches add the desired safety property while preserving the existing properties of input program. However, incremental addition of liveness properties to programs while preserving their existing properties is still an open problem. Also, synthesizing liveness properties is observed as the most difficult part [1, 2, 6] of the existing synthesis methods. Hence, we focus our attention on automatic addition of liveness properties to programs one at a time.

We present a sound and complete synthesis algorithm for adding Leads-to properties to concurrent programs, where a Leads-to property specifies that the program will eventually respond to stimuli of its environment. Specifically, we begin with a desired Leads-to property and the set of transitions of an existing program. Then, we systematically *calculate* the set of transitions of a new program that satisfies the given Leads-to property and the existing specification of the given program. Thus, instead of starting with the specification, we start with the set of transitions of an existing program.

The incremental nature of our approach provides two advantages. First, every time we add a new Leads-to property, our synthesis algorithm has the potential of reusing the computational structure of the input program. As a result, our approach potentially reuses the effort put in the synthesis of one program for the synthesis of its improved version. Second, our synthesis algorithm has the potential to preserve the efficiency of the input program by reusing its computations in the synthesis of its improved version. This aspect is important as the finding in [7] reveals that the inefficiency of automatically generated programs is another obstacle to make synthesis methods practically useful.

As an illustration of our synthesis method, we synthesize a mutual exclusion program consisting of two processes that compete to enter their critical section. We begin with a simple mutual exclusion program that satisfies only its safety property; i.e., it ensures that at most one process is allowed to access the critical section at a time. However, this program does not guarantee progress; i.e., a process may wait forever to enter the critical section. Then, using our synthesis algorithm, we incrementally add the desired Leads-to properties (i.e., each process will eventually get a chance to enter its critical section). We have also used this method to synthesize the readers-writers program with multiple reader processes and a writer process.

Contributions. The contributions of this paper are as follows: (i) we present a sound and complete synthesis algorithm for automatic addition of Leads-to properties to concurrent programs in the high atomicity model – where the processes of programs can atomically read/write all program variables; (ii) we provide potential reuse in the synthesis of concurrent programs where we synthesize an improved version of a program from its existing version, and (iii) we provide a method that is specifically desirable for the maintenance of reactive programs where there is a need for improving program behaviors while providing certain guarantees about the correctness of the improved program.

The organization of the paper. In Section 2, we present preliminary concepts. Then, in Section 3, we formally state the problem of adding Leads-to properties to programs. In Section 4, we present the synthesis algorithm for adding Leads-to properties to programs. We also provide the soundness and the completeness proof of the synthesis algorithm. Subsequently, in Section 5, we synthesize a mutual exclusion program from its input version that does not provide progress for processes. In Section 6, we discuss issues raised by the synthesis method of this paper. Finally, we make concluding remarks and present future work in Section 7.

2 Preliminaries

In this section, we give formal definitions of programs and specifications. Programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [9].

2.1 Program

A program p includes a finite set of variables, say $V = \{v_0, .., v_q\}$, where q is a positive integer, and a finite set of transitions that update these variables. Each variable is associated with a finite domain of values. Let $v_0, .., v_q$ be variables of p, and let $D_0, .., D_q$ be their respective domains. A state of p is obtained by assigning each variable a value from its respective domain. Thus, a state s of p has the form: $\langle l_1, l_2, .., l_q \rangle$ where $\forall i : 1 \leq i \leq q : l_i \in D_i$. The state space of p, S_p, is the set of all possible states of p.

Each program transition is of the form (s_0, s_1) where $s_0, s_1 \in S_p$. Thus, the set of program transitions δ_p is a subset of $S_p \times S_p$. In this paper, in most situations we are interested in the state space of p and all its transitions. Hence, we simply let program p be specified by the tuple $\langle S_p, \delta_p \rangle$, where S_p is a finite set of states and δ_p is a subset of $S_p \times S_p$.

A state predicate of p is any subset of S_p. A state predicate S is closed in the program p (respectively, δ_p) if and only if (iff) $(\forall s_0, s_1 :: ((s_0, s_1) \in \delta_p) \Rightarrow (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$, i.e., a computation prefix need not be maximal.

The projection of program p on state predicate S, denoted as $p|S$, is the following program: $\langle S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \wedge s_0, s_1 \in S\} \rangle$. In other words, $p|S$ consists of transitions of p that start in S and end in S.

Notation. When it is clear from the context, we use p and δ_p interchangeably. Also, we say that a state predicate S is true in a state s iff $s \in S$.

2.2 Specification

A specification is a set of infinite sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [9], we let the specification consist of a safety specification and a liveness specification. For a suffix closed and fusion closed specification, the safety specification can be specified [16] as a set of bad transitions. Thus, for a program p, we represent its safety specification as a set of transitions $spec_{sf}$ that is a subset of $S_p \times S_p$ and p is not allowed to execute any transition in $spec_{sf}$.

Given a program p, a state predicate S, and a specification $spec$, we say that p satisfies $spec$ from S iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true is in $spec$. If p satisfies $spec$ from S and $S \neq \{\}$, we say that S is an invariant of p for spec.

In this paper, we consider a specific class of liveness properties, commonly known as Leads-to properties [17]. Intuitively, a Leads-to property specifies that the program will eventually respond to the stimuli of its environment.

Formally, a computation c satisfies a Leads-to property $L \equiv (R \mapsto Q)$, where R and Q are state predicates, iff it is always the case that if the computation c reaches a state where R holds then c will eventually reach a state where Q holds.

We say that p satisfies $R \mapsto Q$ iff every computation of p satisfies $R \mapsto Q$. In our synthesis algorithms, we require the synthesized program to satisfy the added Leads-to property L and also satisfy its existing Leads-to properties. More specifically, we present the following theorem based on which we define the problem of adding Leads-to properties in Section 3.

Theorem 2.1 Let $p_1 = \langle S_p, \delta_{p_1} \rangle$ and $p_2 = \langle S_p, \delta_{p_2} \rangle$ be two programs with the same state space S_p. If the set of computations of p_1 is a subset of the computations of p_2 then if p_2 satisfies a Leads-to (respectively, a safety) property $\mathcal{P}$ then p_1 satisfies $\mathcal{P}$ as well.

Proof. By contradiction, if a computation c of p_1 violates the Leads-to property $\mathcal{P}$ of p_2 then since c is also a computation of p_2, p_2 will violate $\mathcal{P}$. This is a contradiction. Therefore, p_1 satisfies $\mathcal{P}$. The same argument holds for the case where $\mathcal{P}$ is a safety property. $\square$

Note. We note that Theorem 2.1 holds only for infinite computations of programs. Also, this theorem can only be applied for programs whose liveness specification consists of Leads-to properties. Thus, in the cases where program computations are tree-like structures and liveness is specified in terms of existential quantifications over various paths of execution Theorem 2.1 may not necessarily hold. Therefore, the synthesis algorithm that we present in this paper works for programs whose set of liveness properties is a conjunction of Leads-to properties. Nevertheless, since linear model of computations suffices for specifying the behaviors of a rich class of practical reactive programs (e.g., network protocols) [2], we plan to extend the scope of this work for adding other liveness properties of interest in the linear model of computations (cf. Section 6).

Notation. Let $spec$ be a specification. We use the term safety of $spec$ to mean the smallest safety specification that includes $spec$. Also, whenever the specification is clear from the context, we will omit it; thus, S is an invariant of p abbreviates S is an invariant of p for *spec*.

3 Problem Statement

In this section, we formally define the problem of adding Leads-to properties to programs. Given is a program p, its state space S_p, its specification $spec$, its invariant $S \subseteq S_p$, and a Leads-to property $L \equiv (R \mapsto Q)$, where R and Q are state predicates and $R \not\subseteq Q$. Our goal is to find a program p' with a new invariant S' that satisfies $spec$ and L from S'. We now discuss the conditions for the problem statement based on the requirement that it should be possible to prove that p' preserves certain properties of p based only on the fact that p satisfies those properties.

In the addition of a Leads-to property, we require the calculated invariant S' to be a subset of S. Otherwise, if S' includes states that do not belong to S then p' can start from those states and create new computations that do not belong to p. As a result, we would need to prove that these computations preserve the existing properties of p. Since correctness of p is known only from states in S, we have no knowledge about the computations that start outside S. Hence, based on the above requirement regarding preserving existing properties of p, we require that $S' \subseteq S$.

Also, if the set of transitions of $p'|S'$ includes new transitions that do not belong to $p|S'$ then the set of computations of p' will include new computations where it is not possible to prove the correctness of those new computations based on the correctness of the computations of p. For this reason, we stipulate that the set of transitions $p'|S'$ be a subset of the transitions of $p|S'$ (i.e., $p'|S' \subseteq p|S'$). Hence, we define the problem of adding Leads-to properties to an existing program as follows.

The Problem of Adding Leads-to Properties
Given p, S, $spec$, and the Leads-to property $L \equiv (R \mapsto Q)$
such that p satisfies $spec$ from S, and
R and Q are state predicates,

Identify p' and S' such that

$S \subseteq S'$,
$p'|S' \subseteq p|S'$,
p' satisfies $spec$ from S', and
p' satisfies L from S'. □

Notation. Given an existing program p, specification $spec$, and invariant S, we say that program p' and predicate S' solve the addition problem for a given input iff p' and S' satisfy the three conditions of the addition problem. We say p' (respectively, S') solves the addition problem iff there exists S' (respectively, p') such that p', S' solve the addition problem.

4 Adding Leads-to Properties

In this section, we present a sound and complete algorithm for adding Leads-to properties to programs. In particular, in Subsection 4.1, we present our synthesis algorithm, and then, in Subsection 4.2, we present the soundness and the completeness proof of our algorithm.

4.1 Synthesis Algorithm

In this subsection, we present a synthesis algorithm for solving the addition problem (cf. Section 3). Given is an input program p with its invariant S, its specification $spec$, and a Leads-to property $L \equiv (R \mapsto Q)$, where R and Q are state predicates and $R \not\subseteq Q$. Our goal is to synthesize a program p' with the invariant $S' \subseteq S$ that satisfies $spec$ and L from S'. To achieve our goal, we present the algorithm Add_LeadsTo (cf. Figure 1).

In the implementation of Add_LeadsTo, the notations $\subseteq$, $\cap$, $\cup$, and $\in$ respectively represent the subset, the intersection, the union, and the set membership operators in the set theory. Also, we reuse three auxiliary functions RemoveCycles, RemoveDeadlocks, and EnsureClosure (cf. Figure 3) from [10]. To synthesize the output program p', we consider eight cases based on the possible relations of state predicates R and Q with the invariant S. We show the pictorial representation of these cases in Figure 2.

Cases 1 and 2: Line 1 in Figure 1 (Figures 2-(a, b)). We verify the conditions under which program p inherently satisfies L; i.e., p will never reach a state where R holds (i.e., $R \cap S = \emptyset$) or Q is always satisfied inside the invariant S (i.e., $S \subseteq Q$).

Case 3: Line 2 in Figure 1 (Figure 2-(c)). In the case where the invariant is a subset of R, and Q and S are disjoint, there exists no non-empty subset of S from where computations of p can satisfy L. Thus, it is not possible to add L to program p while preserving the existing properties of p.

Case 4: Lines 3-7 in Figure 1 (Figure 2-(d)). If S is a subset of R and S intersects Q then we have to ensure that starting from every state inside S every computation of p satisfies L. Towards this end, we have to ensure that starting from every state in $(S - Q)$ every computation of p will eventually reach a state in $(S \cap Q)$. For this reason, we use the algorithm RemoveCycles (cf. Figure 3) that takes two state predicates X and Y and the set of transitions of a program p, and guarantees the convergence of the computations of p from every state of X to Y. In other words, it ensures that every computation of p that starts in X will eventually reach a state in Y. Hence, we invoke RemoveCycles with the parameters $(S - Q)$, $(S \cap Q)$, and p in order to ensure the convergence from $(S - Q)$ to $(S \cap Q)$.

The function RemoveCycles (cf. Figure 3) ranks every state $s \in X$ based on the *shortest* path from s to a state in Y. Then, it removes transitions that reach a state from where there is no reachable path to a state in Y. Thus, the invocation of RemoveCycles may create states with no outgoing transitions; i.e., deadlock states. Afterwards, RemoveCycles removes transitions (s_0, s_1) where $s_0 \notin Y$ and the rank of s_1 is greater than or equal to the rank of s_0. Note that in this case the removal of (s_0, s_1) does not create a deadlock state. This is due to the fact that if s_0 were a deadlock state then (s_0, s_1) would be the only outgoing transition of s_0. Hence, since there exists a shortest path from s_0 to Y, the rank of s_1 could not be greater or equal to that of s_0.

```
Add_LeadsTo($S$ : state predicate, $p$: transitions, $R, Q$: state predicate )
{
    If $((R \cap S = \emptyset) \ \vee \ (S \subseteq Q))$   then return $p, S$;                                        (1)
    If $((S \subseteq R) \ \wedge \ (Q \cap S = \emptyset))$   then declare that the addition is not possible; exit();   (2)
    If $((S \subseteq R) \ \wedge \ (Q \cap S \neq \emptyset))$   then
                $p'$ := RemoveCycles($S-Q, S \cap Q, p$);                                                     (3)
                $S'$ := RemoveDeadlocks($S, p'$);                                                            (4)
                If ($S' = \emptyset$) then declare that the addition is not possible; exit();                 (5)
                $p''$ := EnsureClosure($p', S'$);                                                            (6)
                return $p'', S'$;                                                                           (7)
    $p'$ := RemoveCycles($R \cap S, S-R, p$);                                                                (8)
    $S'_{init}$ := RemoveDeadlocks($S, p'$);                                                                 (9)
    If ($S'_{init} = \emptyset$) then declare that the addition is not possible; exit();                     (10)
    If ($S'_{init} \cap R = \emptyset$) then
                $p''$ := EnsureClosure($p', S'_{init}$);                                                     (11)
                return $p'', S'_{init}$;                                                                    (12)
    If ($Q \cap S'_{init} \neq \emptyset$) then
                $Z := \{s : s \in (S'_{init} - ((S'_{init} \cap Q) \cup (S'_{init} \cap R))) \ \wedge$
                                    $s$ is reachable from a state in $(S'_{init} \cap R)\}$;                 (13)
                $p'$ := RemoveCycles($Z, Q \cap S'_{init}, p$);                                              (14)
                $S'$ := RemoveDeadlocks($S'_{init}, p'$);                                                    (15)
                If ($S' = \emptyset$) then declare that the addition is not possible; exit();                 (16)
                $p''$ := EnsureClosure($p', S'$);                                                            (17)
                return $p'', S'$;                                                                           (18)
    declare that the addition is not possible; exit();                                                      (19)
}
```

Figure 1. Adding Leads-to properties.

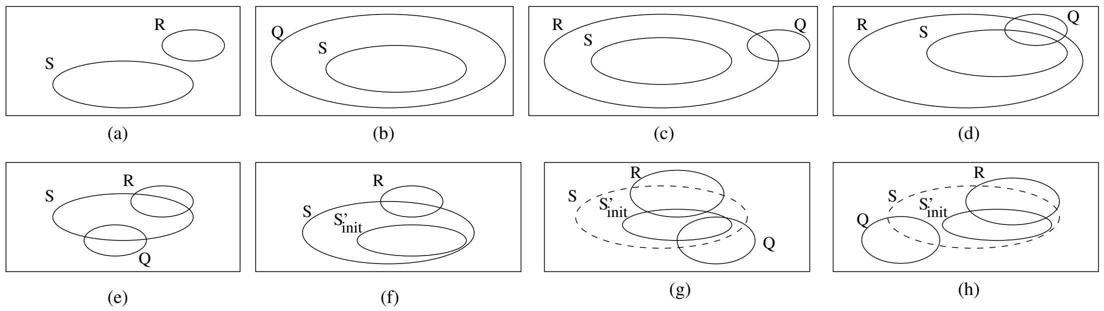

Figure 2. Pictorial representation of Add_LeadsTo algorithm.

Since we require the synthesized program to have infinite computations, we must resolve deadlock states. Thus, in Step 4 (cf. Figure 1), the Add_LeadsTo algorithm invokes the function RemoveDeadlocks (cf. Figure 3) to resolve deadlock states. Since we do not add new transitions in $p'|S'$ (i.e., $p'|S' \subseteq p|S'$), the only option that remains for resolving deadlock states is to make deadlock states unreachable. In other words, we remove deadlock states using RemoveDeadlocks (cf. Figure 3).

After removing deadlock states from S, if the resulting subset S' of S is empty then it will not be possible to add the Leads-to property L to p. Otherwise, we remove transitions that start in S' and end outside S' to ensure the closure of S' as the new invariant. Towards this end, we use the algorithm EnsureClosure (cf. Figure 3). Hence, we return p'' and S' as the synthesized program.

Case 5: Lines 8-10 in Figure 1 (Figure 2-(e)). In the case where S is not a subset of R and the intersection of R and S is non-empty, we ensure that no cycles exist in $R \cap S$ so that every computation of p that reaches $R \cap S$ will eventually be able to exit $R \cap S$. We invoke algorithm RemoveCycles to remove the cycles in $R \cap S$. As we mentioned earlier, when we execute RemoveCycles we need to execute RemoveDeadlocks afterwards because there exists a possibility that RemoveCycles creates deadlock states in the invariant. Hence, we calculate a new

```
RemoveCycles($X, Y$: state predicate, $p$: transitions)
// Ensures the convergence from $X$ to $Y$ using the transitions of $p$.
{
    $\forall s : s \in X : Rank(s) =$ the length of the shortest computation prefix of $p$
                                      that starts from $s$ and ends in a state in $Y$;
    $p := p - \{(s_0, s_1) : Rank(s_1) = \infty\}$;    //$Rank(s) = \infty$ means $Y$ is not reachable from $s$.
    $p' = \{(s_0, s_1) : (s_0, s_1) \in p \wedge (s_0 \in Y \vee Rank(s_0) > Rank(s_1))\}$;
    return $p'$;
}

RemoveDeadlocks($S$ : state predicate, $p$ : transitions)
// Returns the largest subset of $S$ such that computations of $p$ within that subset are infinite
    { while $(\exists s_0 : s_0 \in S : (\forall s_1 : s_1 \in S : (s_0, s_1) \notin p))$ $S := S - \{s_0\}$ }

EnsureClosure($p$ : transitions, $S$ : set of states)
    { return $p - \{(s_0, s_1) : s_0 \in S \wedge s_1 \notin S\}$ }
```

Figure 3. The auxiliary functions used by the Add_LeadsTo algorithm.

invariant S'_{init} after removing cycles in $R \cap S$. If the new invariant S'_{init} is empty then it follows that the removal of cycles in $R \cap S$ culminated in the removal of all invariant states. As a result, the addition would not be possible in such cases.

Case 6: Lines 11-12 in Figure 1 (Figure 2-(f)). If S'_{init} is non-empty and $R \cap S'_{init} = \emptyset$ then this case reduces to Case 1 where the invariant S does not intersect with R. Hence, we will ensure the closure of S'_{init} by calling EnsureClosure and, we return the synthesized program p'' with the invariant S'_{init}.

Case 7: Lines 13-18 in Figure 1 (Figure 2-(g)). If the Add_LeadsTo algorithm reaches Step 13 then $(S'_{init} \cap R \neq \emptyset)$ holds and no cycle exists in $S'_{init} \cap R$ since S'_{init} is the result of removing cycles of $(R \cap S)$ in Step 8. Thus, the only issue that remains is the convergence from $(S'_{init} - ((S'_{init} \cap Q) \cup (S'_{init} \cap R)))$ to $(S'_{init} \cap Q)$. Towards this end, Add_LeadsTo constructs a state predicate Z that is the set of states in $(S'_{init} - ((S'_{init} \cap Q) \cup (S'_{init} \cap R)))$ that are reachable from $(S'_{init} \cap R)$. Afterwards, Add_LeadsTo invokes RemoveCycles in Step 14 to ensure that if a computation starts in $(S'_{init} \cap R)$ and reaches a state in Z then it will eventually reach $(S'_{init} \cap Q)$. Then, it removes deadlock states in order to construct a new invariant S'. If S' is empty then it follows that the removal of cycles in Z has resulted in an empty invariant. Thus, Add_LeadsTo declares that the addition is not possible. Otherwise, it ensures the closure of S' by invoking EnsureClosure, and returns the synthesized program p'' and its invariant S'.

Case 8: Line 19 in Figure 1 (Figure 2-(h)). In the case where S'_{init} and R intersect and Q does not intersect S'_{init}, the addition is not possible because no subset of S'_{init} exists from where computations of p satisfy L.

4.2 Soundness and Completeness of Add_LeadsTo

In this subsection, we show that the algorithm Add_LeadsTo is sound and complete. The algorithm Add_LeadsTo is sound if the synthesized program satisfies the requirements of the addition problem. Also, the algorithm Add_LeadsTo is complete if for a program p, its invariant S, and a LeadsTo property L, Add_LeadsTo succeeds in finding the improved version of p if it exists.

In this subsection, p and S respectively denote the input program and its invariant taken by the Add_LeadsTo algorithm, and p' and S' denote the synthesized program and its invariant. Before we present the proof of soundness, we make the following observations that we use in the proof.

Observation 4.1 The algorithm RemoveDeadlocks returns the largest subset, say S_{sub}, of its argument S, where S_{sub} does not have any deadlock states. □

Observation 4.2 Using Observation 4.1, the state predicate S' returned in Step 7 of Add_LeadsTo is a subset of S. □

Observation 4.3 Using Observation 4.1, the state predicates S'_{init} and S' returned in Steps 12 or 18 of Add_LeadsTo are subsets of S. $\square$

Observation 4.4 The algorithm EnsureClosure does not add any transitions to the set of transitions of its first argument (cf. Figure 3); i.e., program p. $\square$

Observation 4.5 The algorithm RemoveCycles does not add any transitions to the set of transitions of its second argument (cf. Figure 3); i.e., program p. $\square$

Using above observations, we present the following lemmas.

Lemma 4.6 $S' \subseteq S$.

Proof. Based on Observations 4.2 and 4.3, it follows that the predicate S' returned as the invariant of the synthesized program is a subset of S. $\square$

Lemma 4.7 $p'|S' \subseteq p|S'$.

Proof. Using Observations 4.4 and 4.5, we show that the set of transitions $p'|S'$ does not include any new transitions that do not belong to $p|S'$. Therefore, we have $p'|S' \subseteq p|S'$. $\square$

Now, we present the following lemma to show that p' satisfies L from its invariant, S'.

Lemma 4.8 Every computation of p' that starts in R will reach a state in Q.

Proof. We consider two different cases depending on the intersections of R and Q with S'. If $R \cap S' = \emptyset$ then there exists no computation that starts in R. Otherwise, if $R \cap S' \neq \emptyset$ then by construction the reachability of Q from R is guaranteed by removing the reachable cycles and deadlock states in the state predicate $S' - ((S' \cap R) \cup (S' \cap Q))$. Therefore, starting from every state in R, every computation of p will eventually reach a state of Q. $\square$

Now, we show that Add_LeadsTo is sound.

Theorem 4.9 The algorithm Add_LeadsTo is sound.

Proof. To show that Add_LeadsTo is sound, we have to show that the synthesized program p' and its invariant S' satisfy the requirements of the addition problem. Hence, we proceed as follows:

1. $S' \subseteq S$. The proof follows from Lemma 4.6.
2. $p'|S' \subseteq p|S'$. The proof follows from Lemma 4.7.
3. p' satisfies $spec$ from S'. We have $S' \subseteq S$ and $p'|S' \subseteq p|S'$ and S' is non-empty. Also, the EnsureClosure function ensures the closure of S'. Moreover, though EnsureClosure removes a subset of transitions of p' to ensure the closure of S', it does not create any deadlock states in S'. This is due to the fact that in the body of the Add_LeadsTo algorithm we invoke EnsureClosure after RemoveDeadlocks, and RemoveDeadlocks ensures the existence of infinite computations in the new invariant S'. As a result, no state exists in S' from where all outgoing transitions end outside S'. Thus, it follows that the computations of p' are infinite and the set of computations of p' in S' is a subset of the computations of p in S'. Thus, using Theorem 2.1, it follows that p' satisfies $spec$ from S'.
4. p' satisfies L from S'. Using Lemma 4.8, it follows that p' satisfies the Leads-to property $(R \mapsto Q)$ from S'. $\square$

Theorem 4.10 The algorithm Add_LeadsTo is complete.

Proof. Let p'' and S'' solve the addition problem in the state space of p, S_p, and the algorithm Add_LeadsTo fails to find p'' and S''. Since S'' and p'' solve the addition problem, S'' is non-empty and p'' satisfies both $spec$ and $L \equiv (R \mapsto Q)$ from S''. Now, depending on the relationship of R and S'', we consider the following cases.

1. If $S'' \cap R = \emptyset$ then S'' is a subset of S that is closed in the computations of p and does not intersect with R. Furthermore, since p'' satisfies $spec$ from S'', every computation of p'' starting at a state of S'' is an infinite

computation that belongs to $spec$ (i.e., no deadlock state in S''). The algorithm Add_LeadsTo declares that the addition is not possible in Step 10 (cf. Figure 1) if there does not exist such a subset of S that does not intersect with R and has no deadlock state. Otherwise, Add_LeadsTo would have identified S'' either in Step 1 or in Step 12 (cf. Figure 1).

2. For the case that $S'' \cap R \neq \emptyset$, we consider the following cases.
   (a) $S'' \subseteq R$
   Since p'' satisfies $spec$ from S'', every computation c'' of p'' is an infinite computation that belongs to $spec$. Also, since p'' satisfies L from S'' then every computation c'' of p'' must eventually reach a state in $Q \cap S''$. Thus, S'' is a non-empty subset of S from where a subset of computations of p satisfy $spec$ and L; i.e., every computation of p'' that starts in $(S''-Q)$ will eventually reach a state in $(Q \cap S'')$. Based on Observation 4.1, our algorithm calculates the largest subset of S that has the properties of S'' in Steps 3 through 7 (cf. Figure 1). Thus, our algorithm would identify such non-empty subset of S in Steps 3 through 7.
   (b) $S'' \not\subseteq R$
   Every computation c'' of p'' that starts in $R \cap S''$ must eventually reach $Q \cap S''$ (since p'' satisfies L from S''). And, since p'' satisfies $spec$ from S'', S'' is a non-empty subset of S from where a subset of computations of p satisfy $spec$ and L; i.e., every computation of p that starts in $S'' \cap R$ will eventually reach a state in $Q \cap S''$. Therefore, using an argument similar to the previous case, Add_LeadsTo would find S'' and p'' in Steps 13 through 18 (cf. Figure 1).

Therefore, if there exists a program p'' and its invariant S'' that solves the addition problem then the Add_LeadsTo will succeed in finding p'' and S''. □

5 Examples

In this section, we present the synthesis of two programs using our synthesis algorithm. First, in Section 5.1, we present the synthesis of a mutual exclusion program with two processes. Second, in Section 5.2, we synthesize the readers-writers program with two reader processes and a writer process.

5.1 Mutual Exclusion Program

In this section, we present an example for adding Leads-to properties to an existing program. We use guarded commands as shorthand for representing the set of program transitions. A guarded command $g \rightarrow st$ captures the transitions $\{(s_0, s_1)$: the state predicate g is true in s_0, and s_1 is obtained by atomic *execution* of statement st in state s_0 $\}$. Also, we represent each individual process by a subset of the set of program transitions.

The mutual exclusion program, ME. The initial mutual exclusion program consists of two processes P_0 and P_1 that share a critical section. Each process P_j, for $0 \leq j \leq 1$, has three Boolean variables n_j, t_j, and c_j, where (i) n_j shows that P_j is in its non-critical section; (ii) t_j shows that P_j intends to enter its critical section (i.e., trying section), and (iii) c_j shows that P_j is in its critical section.

If a process P_j is in its non-critical section then P_j can enter its trying section (cf. action $ME1_j$). When P_j is in its trying section and the other process (i.e., $P_{(j\oplus 1)}$, where $\oplus$ denotes modulo 2 addition) is not in its critical section $c_{(j\oplus 1)}$ then P_j can enter its critical section. Finally, P_j can exit its critical section, c_j, and enter its non-critical section, n_j. Hence, we represent the actions of each process P_j as follows (for $j = 0, 1$):

$$\begin{array}{lll} ME1_j: & n_j & \longrightarrow & t_j := true; n_j := false; \\ ME2_j: & t_j \wedge \neg c_{(j\oplus 1)} & \longrightarrow & c_j := true; t_j := false; \\ ME3_j: & c_j & \longrightarrow & n_j := true; c_j := false; \end{array}$$

Invariant. The following state predicate is an invariant of ME.

$$S_{ME} = (\forall j : 0 \leq j \leq 1 : (n_j \wedge \neg t_j \wedge \neg c_j) \vee (t_j \wedge \neg n_j \wedge \neg c_j) \vee (c_j \wedge \neg t_j \wedge \neg n_j)) \;\wedge\; \neg(c_0 \wedge c_1)$$

Safety specification. The safety specification of ME requires the program not to reach states s_1 where both processes P_0 and P_1 are in their critical sections; i.e., $c_0(s_1)$ and $c_1(s_1)$ are true where $c_0(s_1)$ (respectively, $c_1(s_1)$) denotes the value of c_0 (respectively, c_1) at s_1. Hence, we represent the safety specification of ME as the following set of transitions that ME is not allowed to execute.

$$sf_{ME} = \{(s_0, s_1) : c_0(s_1) \wedge c_1(s_1)\}$$

Now, for the ease of presentation, we illustrate the reachability graph of the initial program in Figure 4 that shows all reachable states from the initial state s_i where both processes are in there non-critical sections; i.e., $n_0(s_i)$ and $n_1(s_i)$ hold. We have annotated each transition with the index of the process that executes that transition.

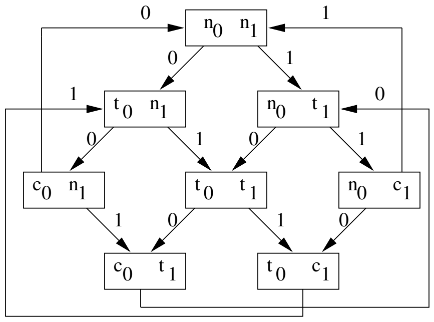

Figure 4. The reachability graph of program ME.

The desired Leads-to property. Program ME satisfies its safety specification but there exists a possibility of starvation for either one of processes; i.e., there is no guarantee for a process that is waiting in its trying section to enter its critical section. Hence, we add the progress property $(R \mapsto Q)$ to program ME, where $R \equiv t_j$ and $Q \equiv c_j$ $(0 \leq j \leq 1)$, for both processes P_0 and P_1. Next, we trace our algorithm to add progress for P_0 (i.e., $j = 0$). Since the structures of the processes P_0 and P_1 are symmetric, we omit the addition of progress for P_1 and only present the synthesized program.

Adding the Leads-to property. We trace the execution of our algorithm for adding the Leads-to property $(R \mapsto Q)$ to program ME, where $R \equiv t_0$ and $Q \equiv c_0$.

Since $R \cap S_{ME} \neq \emptyset$ and S_{ME} is not a subset of Q, the first condition in the Add_LeadsTo (cf. Line 1 in Figure 1) does not hold for ME program. Also, since $Q \cap S_{ME} \neq \emptyset$, the Add_LeadsTo algorithm moves to Step 2. The execution of Add_LeadsTo continues from Step 8 (cf. Figure 1) since the invariant S_{ME} is not a subset of R. The intersection of R and S_{ME} is equal to the following state predicate.

$$R \cap S_{ME} = (t_0 \wedge \neg n_0 \wedge \neg c_0) \wedge ((n_1 \wedge \neg t_1 \wedge \neg c_1) \vee (t_1 \wedge \neg n_1 \wedge \neg c_1) \vee (c_1 \wedge \neg t_1 \wedge \neg n_1))$$

After removing a cycle in $R \cap S_{ME}$ and calculating a new invariant S'_{init}, the new invariant becomes equal to S_{ME}. Since $R \cap S'_{init}$ is non-empty, the execution of the Add_LeadsTo algorithm continues from Step 13. Since no more cycles exist in $R \cap S'_{init}$, we construct the transitions of the synthesized program and exit at Step 18. Afterwards, we execute Add_LeadsTo one more time to add the property $(t_1 \mapsto c_1)$. Finally, the actions of the synthesized program for a process P_j are as follows $(j = 0, 1)$.

$$\begin{array}{lll} ME1'_j: & n_j \wedge \neg t_{(j \oplus 1)} & \longrightarrow \; t_j := true; n_j := false; \\ ME2'_j: & t_j \wedge n_{(j \oplus 1)} & \longrightarrow \; c_j := true; t_j := false; \\ ME3'_j: & c_j & \longrightarrow \; n_j := true; c_j := false; \end{array}$$

Also, we show the reachability graph of program ME' in Figure 5.

One could argue that the above program does not allow a process to move to its trying section from its non-critical section, and as a result, a process may be starved at its non-critical section. In order to remedy this shortcoming of the program ME', we add the Leads-to properties $(n_j \mapsto t_j)$ to ME' (for $j = 0, 1$). The actions of the synthesized program for a process P_j are as follows.

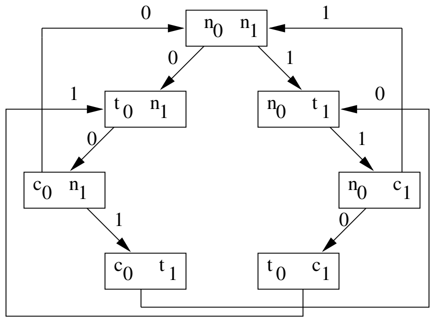

Figure 5. The reachability graph of program ME'.

$$\begin{array}{lll} ME1''_j: & n_j \wedge \neg t_{(j\oplus 1)} & \longrightarrow \quad t_j := true; n_j := false; \\ ME2''_j: & t_j \wedge \neg c_{(j\oplus 1)} & \longrightarrow \quad c_j := true; t_j := false; \\ ME3''_j: & c_j \wedge t_{(j\oplus 1)} & \longrightarrow \quad n_j := true; c_j := false; \end{array}$$

The above program satisfies the safety specification sf_{ME} and Leads-to properties $(t_j \mapsto c_j)$ and $(n_j \mapsto t_j)$ $(j = 0, 1)$ from the invariant S'_{ME}, where

$$S'_{ME} = S_{ME} - \{s : (t_0(s) \wedge t_1(s)) \vee (n_0(s) \wedge n_1(s))\}$$

We show the reachability graph of this program in Figure 6. Although the reachability graph of program ME'' reflects an alternation between processes for entering the critical section, the synthesized program behaves as a token passing mutual exclusion program. In other words, when a process is in the critical section, it can hold the token and remain in the critical section as long as the other process has not requested the token. When the other process requests the token, the token is passed to it and then that process can enter the critical section. The same argument holds for the case where the speeds of processes are different.

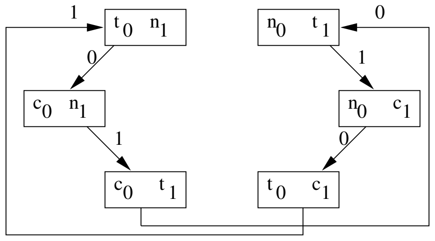

Figure 6. The reachability graph of a token passing mutual exclusion program ME''.

5.2 Readers-Writers Program

In this section, we present the addition of Leads-to properties to a version of the readers-writers problem that consists of two reader processes, a writer process at each moment, a process that manages an internal queue of writers. We first describe the initial program to which we add Leads-to properties. Subsequently, we present the program that we have synthesized using our algorithm.

The readers-writers program, RW. Multiple writer processes wait in an infinite external queue to be pick by the readers-writers program, RW. The RW program contains a finite internal queue with size 2 that is managed by a *queue manager* process. The queue manager process manages the selection of writers from external queue and places them in the internal queue. The selected writer has access to a common buffer to which two reader processes have access as well. At any time there exists only one writer that is allowed to write the buffer. Also, two other processes R_0 and R_1 read the data from the common buffer. The program has three integer variables $0 \leq nr \leq 2$, $0 \leq nw \leq 1$, and $0 \leq nq \leq 2$ that are initially 0 where nr represents the number of readers reading from the buffer, nw represents the number of writers writing the buffer, and nq represents the number of writers waiting in the internal queue. Also, the program contains Boolean variables rd_0, rd_1, and wrq. When R_0

(respectively, R_1) is reading the buffer the value of rd_0 (respectively, rd_1) is $true$. Also, the variable wrq becomes $true$ when there exists at least a writer waiting in the internal queue. The variable wrq is set to $true$ by the queue manager when there is a process waiting to write and wrq is set to $false$ by the program when a process is writing the buffer.

Safety specification. The safety specification of the program requires that when a writer is writing in the buffer no other process is allowed to access the buffer. However, multiple readers can read the buffer simultaneously. Thus, we represent the safety specification as follows:

$$sf_{RW} = \{(s_0, s_1) : (nw(s_1) > 1) \vee ((nr(s_1) \neq 0) \wedge nw(s_1) \neq 0))\}$$

The safety specification stipulates that the condition $(nw \leq 1) \ \wedge \ ((nr = 0) \vee nw = 0))$ must hold in every reachable state. Another representation of the above formula is $0 \leq (3-(nr+3 \cdot nw))$, where 3 is the total number of readers and writers; i.e., $(nw \leq 1) \ \wedge \ ((nr = 0) \vee nw = 0))$ holds if and only if $0 \leq (3-(nr+3 \cdot nw))$ holds. For ease of presentation, we represent the condition $(3-(nr + 3 \cdot nw))$ with a variable K.

Invariant. The state predicate S_{RW} represents the invariant of the program, where

$$S_{RW} = (0 \leq K) \ \wedge \ ((nr = 0) \Rightarrow (\neg rd_1 \wedge \neg rd_2)) \ \wedge \\ ((nr = 1) \Rightarrow (rd_1 \overline{\vee} rd_2)) \ \wedge \ ((nr = 2) \Rightarrow (rd_1 \wedge rd_2))$$

(The operator $\overline{\vee}$ represents the logical XOR.)

The actions of the initial program, RW**.** The actions of the writer process in the initial program are as follows:

$$\begin{array}{lll} W_1 : \ (nq > 0) \wedge (3 \leq K) & \longrightarrow & nw := nw + 1; nq := nq - 1; wrq := false; \\ W_2 : \ (nw = 1) & \longrightarrow & nw := nw - 1; \end{array}$$

When there exists a process ready for writing in the internal queue (i.e., $nq > 0$) and no reader is reading the common buffer (i.e., $3 \leq K$) the program allows the writers to write the common buffer. Thus, the writer process waits until all readers finish their reading activities. When a writer process accesses the buffer, it increments the value of nw, sets the value of wrq to $false$, and decrements the value of nq in order to allow the queue manager to allow other waiting writers in (cf. Action W_1). When the writer finishes its writing activity in the buffer, it exists by decrementing the value of nw (cf. Action W_1).

We represent the actions of the readers as parameterized actions since the structures of the readers are symmetric ($j = 0, 1$).

$$\begin{array}{lll} R_{j_1} : \ \neg wrq \wedge \neg rd_j \wedge (1 \leq K) & \longrightarrow & nr := nr + 1; rd_j := true; \\ R_{j_2} : \ rd_j & \longrightarrow & nr := nr - 1; rd_j := false; \end{array}$$

The value of condition K is greater or equal to 1 if and only if there exists no writer process writing in the buffer. Thus, if a reader process is not already in reading status and no writer is writing in the buffer (cf. Action R_{j_1}) then the reader can read the buffer. When a reader process R_j ($j = 0, 1$) finishes its reading activity, it decrements the value of nr and sets rd_j to $false$. Now, we present the action of the queue manager process.

$$QM : \ (nq < 2) \ \longrightarrow \ nq := nq + 1; wrq := true;$$

Once the queue manager selects a waiting writer, it increments the value of nq and sets the value of wrq to true in order to show that a writer is waiting in the internal queue.

The desired Leads-to property. The initial program satisfies the safety specification sf_{RW}, however, there exists no guarantee for the readers processes and the writer process to access the buffer; i.e., no progress is guaranteed. For example, the writer process may wait forever due to alternating access of R_0 and R_1 to the buffer. The desired Leads-to property for reader R_0 is $R \mapsto Q$, where $R \equiv (0 \leq K)$ and $Q \equiv (rd_0)$. Likewise, we require the

Leads-to property $(0 \leq K) \mapsto rd_1$ for reader process R_1. For the writer process, we require the Leads-to property $R \mapsto Q$, where $R \equiv (nq > 0)$ and $Q \equiv (nw = 1)$; i.e., if there exists a writer process waiting in the queue then it will eventually write the buffer.

Adding the Leads-to property. We apply our algorithm incrementally in order to add the above Leads-to properties to the initial program. The actions of the writer process are as follows:

$$\begin{array}{lll} W_1' : & (wrq) \wedge (K = 3) & \longrightarrow & nw := nw + 1; nq := nq - 1; wrq := false; \\ W_2' : & (\neg wrq) \wedge (K = 0) & \longrightarrow & nw := nw - 1; \end{array}$$

Since the structures of the synthesized reader processes are symmetric, we use parameterized actions to represent the transitions of the reader processes ($j = 0, 1$).

$$\begin{array}{lll} R_{j_1}' : & (\neg wrq) \wedge \neg rd_j \wedge (1 < K) & \longrightarrow & nr := nr + 1; rd_j := true; \\ R_{j_2}' : & rd_j \wedge (wrq) \wedge (K < 3) & \longrightarrow & nr := nr - 1; rd_j := false; \end{array}$$

The queue manager process should also modify its behavior in selecting writers and placing them in the internal queue:

$$QM' : \ (nq = 0) \wedge (K = 1) \ \longrightarrow \ nq := nq + 1; wrq := true;$$

6 Discussion

In this section, we address some questions raised about the synthesis method presented in this paper. Specifically, we discuss related work and the applications of our synthesis algorithm.

How does the approach of this paper differ from existing synthesis methods in the literature?

We address this question with respect to three different synthesis approaches in the literature.

- *Synthesizing synchronization skeleton of programs from temporal logic specifications.*

  The synthesis method in this approach [1, 2, 5, 6, 18] is based on a decision procedure for satisfiability proof of the specification. Although such synthesis methods may have slight differences with respect to the input specification language and the program model that they synthesize, the general approach is based on the satisfiability proof of the specification. This issue makes it difficult to provide reuse in the synthesis of programs; i.e., any changes in the specification require the synthesis to be restarted from scratch. By contrast, since the input to our synthesis method is the set of transitions of a program, our approach has the potential to reuse those transitions in the synthesis of an improved version of the input program.

  Nevertheless, similar to the above-mentioned methods that generate the synchronization skeleton (i.e., abstract structure) of programs, we also generate the abstract structure of programs. Synthesizing the abstract structure of programs allows us to (i) focus on concurrency issues in the synthesis of concurrent and distributed programs instead of their functional properties, and (ii) provide the potential of translating the abstract structure of the synthesized program to multiple programming languages unlike approaches that focus on the synthesis of programs in a specific programming language [8].

- *Synthesizing proof-carrying (certified) code.* In this approach, the synthesis method takes the input specification and generates the code of the program annotated by its proof of correctness [12, 19]. Then, a proof checker verifies the correctness of the synthesized program to provide high assurance in the synthesis of safety-critical systems. Also, in the synthesis of certified code, there exists an option for adding domain-specific knowledge in order to derive more efficient programs. However, such approaches mostly focus on safety properties of programs whereas our focus is to add new liveness properties to programs.

- *Synthesizing fault-tolerant programs.* Our synthesis method differs from the algorithms for automatic addition of fault-tolerance to programs [10, 11, 20–22] in that we do not deal with faults in the synthesis algorithm presented in this paper. Specifically, since in the presence of fault transitions the state of the program can be perturbed outside the invariant, the algorithms in [10] deal with the addition of safety and liveness even from the states outside the invariant, whereas our algorithm only deals with adding liveness in the scope of the invariant of the input program.

Why did we choose Leads-to *properties?*

Our choice of Leads-to properties has a practical bias behind it, as the liveness properties of reactive programs (e.g., network protocols) are often specified in terms of Leads-to properties. In principle, Leads-to properties model the response of reactive programs to the stimuli of their environment. Specifically, corresponding to events happened in the environment of a reactive program, the program must reply with an appropriate response so that the specification of the program is satisfied. In addition to Leads-to properties, we are investigating the automatic addition of other properties of Linear Temporal Logic [23] (e.g., until properties denoted $(R \,\mathcal{U}\, Q)$) to programs based on which we plan to develop a synthesis tool for incremental synthesis of reactive programs.

What will happen if the new Leads-to *properties have new variables?*

In the cases where the desired Leads-to property introduces new variables to the set of program variables, we present the following approach for automatic addition of the given property. We expand the state space by defining a mapping $H : S_{p_o} \rightarrow S_{p_n}$, where S_{p_o} and S_{p_n} are respectively the old and the new state spaces. Using H, we calculate the images of every state $s \in S_{p_o}$ that is a set of states $s' \in S_{p_n}$.

Also, using mapping H, we construct a set of transitions (s'_0, s'_1) in the new state space S_{p_n} corresponding to every transition (s_0, s_1) of the input program p_o, where the values of new variables are the same at s'_0 and s'_1. Thus, we construct the set of transitions of p_n in S_{p_n}. Likewise, we use H to map state predicates R_o, Q_o, and S_o in S_{p_o} to state predicate R_n, Q_n, and S_n in S_{p_n}. Afterwards, we invoke the Add_LeadsTo algorithm on the program p_n and state predicates R_n, Q_n, and S_n in the new state space S_{p_n}.

Where does the approach of this paper apply in the development of software systems?

In the development of software systems, it is often the case that developers have incomplete knowledge about the problem at hand. As a result, the initial specification of programs is often incomplete. Thus, as developers learn new aspects of a problem they add new properties to the program specification. Also, in the maintenance of software systems, developers often add new properties to programs in order to address issues that have not been considered in the design of the software system.

Our incremental approach allows developers to easily add new properties to programs while reusing the existing program. Since we automatically add new properties to a given program, the resulting program will be correct by construction; i.e., the synthesized program guarantees to satisfy the old specification and the newly added property. Such an incremental approach allows us to reuse the effort put in the design of programs as we improve their behaviors.

Moreover, our synthesis algorithm has the potential to be integrated in modeling environments (e.g., UML modeling environments) in order to generate a realization of the model for developers. More specifically, using existing approaches [24] that generate formal specification of models, our synthesis algorithm can automatically synthesize corresponding program that realizes the original model.

How important is the choice of the initial program?

Our algorithm takes the initial program p and a Leads-to property $\mathcal{P}$ and adds $\mathcal{P}$ to p if possible. Thus, as far as the properties of our synthesis algorithm are concerned the answer to this question is out of the scope of this paper. However, the choice of the initial program can affect the result of addition. Specifically, if we start with an initial program that is maximal, i.e., has the weakest invariant and the maximum non-determinism then the chance of a successful addition is higher. Also, based on the Observation 4.1, our algorithm finds the weakest invariant and the maximal program as well. This issue is particularly important for an incremental addition of Leads-to properties where the synthesized program for one Leads-to property is the input to our algorithm for the addition of another Leads-to property (e.g., Section 5).

How does the synthesis algorithm preserve the properties of the input program whereas it removes some computations?

Although the synthesis algorithm removes a subset of computations of the input program, based on Theorem 2.1, the synthesized program includes a subset of the infinite computations of the input program that satisfy its existing Leads-to (respectively, safety) properties.

7 Conclusion and Future Work

We presented a sound and complete synthesis algorithm for automatic addition of Leads-to properties to concurrent programs. Such addition is desirable as the synthesized program is correct by construction. Also, our synthesis method has the potential to provide an automatic approach for the maintenance of reactive programs where we allow developers to systematically add new properties to programs while reusing the existing program. Such incremental approach is also important in the specification of software systems where developers add new properties to program specification as they learn new aspects of the problem at hand. Moreover, our synthesis algorithm extends the scope of program synthesis in the sense that we provide reuse in the incremental synthesis of concurrent programs. As an illustration of our synthesis method, we synthesized a mutual exclusion program by incremental addition of progress properties to a program that satisfies only the safety properties of mutual exclusion problem. Also, we have synthesized a readers-writers program with multiple reader processes and a writer process.

Our synthesis algorithm differs from the cases where one synthesizes a program from its temporal logic specification [1, 2, 4–7, 25–27]. We start with the set of transitions of a program and incrementally add new liveness properties to the program at hand, whereas in specification-based approaches one starts with the specification and synthesizes the program from the satisfiability proof of the specification. As a result, if one adds a new property to the specification then one will have to re-synthesize the program from scratch. In our approach, we systematically add new liveness properties to programs with the potential of reusing the computational structure of the program during synthesis.

We plan to develop a synthesis tool where developers can automatically add new liveness properties to existing programs. Towards this end, in addition to the algorithm presented in this paper, we are investigating the automatic addition of until properties (denoted $R\,\mathcal{U}\,Q$) [23] to concurrent programs. Also, we would like to investigate the automatic addition of liveness properties where we have non-linear computational model (e.g., tree-like structures). Yet another extension to this work is the design of synthesis algorithms for adding liveness properties to distributed programs considering distribution issues.

References

[1] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[2] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[3] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesis. *In Proc. Of 31st IEEE Symposium on Foundation of Computer Science*, pages 746–757, 1990.

[4] P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

[5] P. Attie. Synthesis of large concurrent programs via pairwise composition. *CONCUR'99: 10th International Conference on Concurrency Theory*, 1999.

[6] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

[7] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[8] Xinghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis and verification of synchronization in concurrent programs. *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

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

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

[11] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *Proceedings of the 22nd International Conference on Distributed Computing Systems*, page 337, 2002.

[12] Bernd Fisher, Johann Schumann, and Mike Whalen. Synthesizing certified code. *In Proceedings Formal Methods Europe(FME'02). Copenhagen, Denmark. LNAI, Springer*, 2002.

[13] Ewen Denney, Bernd Fischer, and Johann Schumann. Adding assurance to automatically generated code. *In Proceedings the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE 2004). Tampa, Florida*, March 2004.

[14] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. *In Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), volume 2280 of Lecture Notes in Computer Science*, pages 342–356, October 2002.

[15] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. *In ACM SIGSOFT Conference on the Foundations of Software Engineering /European Software Engineering Conference, Helsinki, Finland*, pages 337–346, 2003.

[16] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[17] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[18] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

[19] A.W. Appel and A.P. Felty. A semantic model of types and machine instructions for proof-carrying code. *In Proceedings of the 27th ACM Symposium of Principles of Programming Languages, ACM Press*, pages 243–253, 2001.

[20] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, page 130, 2001.

[21] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 441, 2003.

[22] Sandeep S. Kulkarni and Ali Ebnenasir. Automated synthesis of multitolerance. *To appear in Proceedings the International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy*, June 28 - July 1 2004.

[23] E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[24] William E. McUmber and Betty H.C. Cheng. Generic framework for formalizing UML. *In Proceedings of IEEE International Conference on Software Engineering (ICSE01), Toronto, Canada.*, 2001.

[25] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specification. *In Proceeding of 16th International Colloqium on Automata, Languages, and Programming*, 1989.

[26] A. Pnueli and R. Rosner. On the synthesis of a reactive module. *In Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[27] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. *In Proceeding of 16th International Colloqium on Automata, Languages, and Programming*, Lec. Notes in Computer Science 372, Springer-Verlag:652–671, 1989.