Revising UNITY Programs: Possibilities and Limitations[1]

Ali Ebnenasir, Sandeep S. Kulkarni, and Borzoo Bonakdarpour

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering,
Michigan State University,
48824 East Lansing, Michigan, USA
{ebnenasi, sandeep, borzoo}@cse.msu.edu
http://www.cse.msu.edu/~{ebnenasi,sandeep,borzoo}

Abstract. We concentrate on automatic addition of UNITY properties unless, stable, invariant, and leads-to to programs. We formally define the problem of adding UNITY properties to programs while preserving their existing properties. For the cases where one simultaneously adds a single leads-to property along with a conjunction of unless, stable, and invariant properties to an existing program, we present a sound and complete algorithm with polynomial time complexity (in program state space). However, for the cases where one simultaneously adds more than one leads-to properties to a program, we present a somewhat unexpected result that such addition is NP-complete. Therefore, in general, adding one leads-to property is significantly easier than adding two (or more) leads-to properties.

Keywords: UNITY, Formal Methods, Program Synthesis

1 Introduction

In this paper, we focus on automated addition of UNITY properties [1] to existing programs. To motivate the application of this work, consider two scenarios: In the first scenario, *a designer checks a program to determine if it satisfies the given properties of interest using a model checker. The model checker provides a counterexample demonstrating that one of the properties is not met.* In this scenario, the designer needs to modify the given model so that it satisfies that property (while ensuring that the remaining properties continue to be satisfied). In another scenario, an existing program needs to be modified so that it satisfies an additional property of interest (while satisfying existing properties). Such a scenario occurs when the specification is incomplete and as designers gain more domain knowledge about the problem at hand, they may add new properties to the specification.

[1] 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.

There exist two ways in which one can deal with the above scenarios: (1) *local redesign*, where the designer removes the program behaviors that violate the property of interest without adding any new behaviors, or (2) *comprehensive redesign*, where the designer introduces new behaviors in the program computations (e.g., by introducing new variables, or adding new computation paths). Clearly, the former approach is desirable, as it ensures that certain existing specifications (e.g., the UNITY specifications from [1]) are preserved. Moreover, in the second scenario, the designer may not have access to the complete specification of the existing system. Hence, in this case, local redesign, if successful, is highly desirable.

We expect that an algorithm for local redesign would be especially useful if it were sound and complete. A sound algorithm ensures that the redesigned program meets the new specification (in addition to preserving existing specification); i.e., the redesigned program is correct by construction. Moreover, a complete algorithm provides an insight for the designer to decide if a program can be redesigned locally or it should be redesigned from scratch to satisfy a new property while preserving its exiting properties. Such automated assistance for the designer is highly desirable since it significantly decreases the design time by warning the designers about spending time on fixing a program that is not *fixable*.

With this motivation, we present an incremental method for adding UNITY properties to programs. Our incremental approach has the potential to reuse the computations of an existing program while adding new properties to it. Also, we focus on UNITY since it provides (i) a simple and general computational model for a variety of computing systems, and (ii) a proof system for refining programs [1]. We expect to benefit from simplicity and generality of UNITY in automatic design of programs.

The basic UNITY properties from [1] are *unless, stable, invariant, ensures*, and *leads-to*. (We refer the reader to Section 2 for precise definitions.) Of these, *ensures* can be expressed in terms of *leads-to* and *unless*. Hence, we focus on adding *unless, stable, invariant*, and *leads-to* to programs. In particular, we present a sound and complete algorithm for simultaneous addition of a single *leads-to* property and a conjunction of *unless, stable*, and *invariant* properties. The time complexity of our algorithm is polynomial in program state space. However, we present an unexpected result that simultaneous addition of more than one *leads-to* property to a program is NP-complete. Based on this result, we find that adding one *leads-to* property is significantly easier than simultaneous addition of two *leads-to* properties.

Contributions. The contributions of this paper are as follows: (1) We formally define the problem of adding UNITY properties to programs; (2) We present a sound and complete algorithm for automatic addition of a *leads-to* property and a conjunction of *unless, stable*, and *invariant* properties to programs, and (3) We show that simultaneous addition of more than one *leads-to* property to a program is NP-complete.

Organization of the paper. First, we present preliminary concepts in Section 2. In Section 3, we formally define the problem of adding UNITY properties to programs. Then, in Section 4, we present our sound and complete algorithm for adding a *leads-to* property to programs. In Section 5, we present our NP-completeness result. Subsequently, in Section 6, we demonstrate our addition algorithm using a mutual exclusion program. In Section 7, we compare the results of this paper with related work. We discuss the limitations and the applications of our results in Section 8. Finally, we make concluding remarks in Section 9.

2 Preliminaries

In this section, we give formal definitions of programs and properties in UNITY [1]. Programs are defined in terms of their state space and their transitions. UNITY properties are defined in terms of infinite sequences of transitions.

Program. A program p is of the form $\langle S_p, I_p, \delta_p \rangle$, where S_p is a finite set of states, $I_p \subseteq S_p$ is the set of initial states of p, and $\delta_p \subseteq S_p \times S_p$ is the set of transitions of p.

A state predicate of p is any subset of S_p. A sequence of states, $\sigma = \langle s_0, s_1, \cdots \rangle$ is a computation of p iff (if and only if) the following three conditions are satisfied: (1) $s_0 \in I_p$; (2) if σ is infinite then $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$ holds, and (3) if σ is finite and terminates in state s_f then there does not exist state s such that $(s_f, s) \in \delta_p$, and $\forall j : 0 < j \leq f : (s_{j-1}, s_j) \in \delta_p$ holds. A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$.

Properties of UNITY Programs. We reiterate the definition of the UNITY properties from [1]. In the following definitions, P and Q are state predicates.

- *Unless.* An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies P *unless* Q iff $\forall i : 0 \leq i : (s_i \in (P \cap \neg Q)) \Rightarrow (s_{i+1} \in (P \cup Q))$. Intuitively, the sequence σ satisfies P *unless* Q iff if P holds in some state of σ then either (1) Q never holds in σ and P is continuously true, or (2) Q eventually becomes true and P holds at least until Q becomes true.
- *Stable.* An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies $stable(P)$ iff σ satisfies (P *unless* $false$). Intuitively, P is stable iff once it becomes true it remains true forever.
- *Invariant.* An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies $invariant(P)$ iff $s_0 \in P$ and σ satisfies $stable(P)$. An invariant property always holds.
- *Ensure.* An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies P *ensures* Q iff (σ satisfies P *unless* Q) and $(\exists j : 0 \leq j : s_j \in Q)$. In other words, there exists a state s_j , where (i) Q eventually becomes true in s_j , and (ii) P remains true everywhere between the first state s_i, $i \leq j$, where P becomes true and s_j.
- *Leads-to* (denoted $\mapsto$). An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies $P \mapsto Q$ iff $(\forall i : 0 \leq i : (s_i \in P) \Rightarrow (\exists j : i \leq j : s_j \in Q))$. If P holds in some state $s_i \in \sigma$ then there exists a state $s_j \in \sigma$ where Q holds and $i \leq j$.

Since *ensures* can be expressed as a conjunction of an *unless* property and a *leads-to* property, we do not consider it explicitly in this paper. Now, let *spec* be any conjunction of the above properties; i.e., $spec = \mathcal{L}_1 \wedge \cdots \wedge \mathcal{L}_n$, where $\mathcal{L}_i$ belongs to the set of properties *unless, stable, invariant*, and *leads-to* $(1 \leq i \leq n)$. A sequence of states $\sigma = \langle s_0, s_1, ...\rangle$ satisfies *spec* iff $\forall i : 1 \leq i \leq n : \sigma$ satisfies $\mathcal{L}_i$.

The properties *unless, stable*, and *invariant* are safety properties, as defined by Alpern and Schneider [2]. These properties can be modeled in terms of a set of *bad* transitions that should never occur in a program computation. For example, $stable(P)$, requires that transitions of the form (s_0, s_1), where $s_0 \in P$ and $s_1 \notin P$ should never occur in any program computation. Hence, for simplicity, in this paper, when dealing with these properties, we assume that they are represented as a set of *bad* transitions $\mathcal{B} \subseteq S_p \times S_p$ that should not occur in any computation.

Finally, we say that program p satisfies a given UNITY specification, *spec*, iff all computations of p are infinite and every computation of p satisfies *spec*.

Remark. We distinguish between a terminating computation and a deadlocked computation. To model a computation that terminates in state s_f, we include the transition (s_f, s_f) in program p. When a computation c of p reaches s_f, c can be extended to an infinite computation by stuttering at s_f. If there exists a state s_d such that there is no outgoing program transition from s_d then s_d is a deadlocked state and a computation of p that reaches s_d is a deadlocked computation. Such computations cannot be extended to an infinite computation. We want to ensure that such deadlocked computations do not occur while revising a program.

3 Problem Statement

In this section, we formally define the problem of adding UNITY specifications to programs. Given is a program p (with state space S_p, initial states I_p and transitions δ_p) that satisfies a UNITY specification $spec_e$. The goal is to generate a modified version of p, denoted p', in such a way that p' satisfies a UNITY specification $spec_n$, (in addition to preserving its existing specification $spec_e$). Moreover, this addition should be done in such a way that one does not need to know the existing specification $spec_e$; during the addition, we only want to reuse the correctness of p with respect to $spec_e$ so that the correctness of p' with respect to $spec_e$ is derived from 'p satisfies $spec_e$'.

Now, we identify constraints on $S_{p'}$, $I_{p'}$ and $\delta_{p'}$. Clearly, in obtaining $S_{p'}$, no new states should be added to S_p; otherwise, there is no guarantee that the correctness of p can be reused to ensure that existing specification will continue to be preserved. Moreover, since S_p denotes the set of all states (not just reachable states) of p, removing states from S_p is not advantageous. Likewise, $I_{p'}$ should not have any states that were not there in I_p. Moreover, since I_p denotes the set of all initial states of p, we should preserve them during the transformation. Finally, likewise, $\delta_{p'}$ should be a subset of δ_p. Note that not all transitions of δ_p may be preserved in p'. However, we must ensure that p' does not deadlock in any reachable state. Based on the definition of the UNITY specification, if (i) $\delta_{p'} \subseteq \delta_p$, (ii) p' does not deadlock in any reachable state, and (iii) p satisfies $spec_e$, then p' also satisfies $spec_e$. Thus, the problem statement is defined as follows:

The Problem of Adding UNITY Properties
Given a program p, its state space S_p, its set of initial states I_p, and a UNITY specification $spec_n$, identify
$\delta_{p'}$, $S_{p'}$, and $I_{p'}$ such that

$(C1)$ $S_{p'} = S_p$
$(C2)$ $I_{p'} = I_p$
$(C3)$ $\delta_{p'} \subseteq \delta_p$
$(C4)$ p' satisfies $spec_n$ □

Note that the requirement of deadlock freedom is not explicitly specified in the above problem statement, as it follows from 'p' satisfies $spec_n$'.

4 Adding Single Leads-to and Multiple Safety Properties

In this section, we present a simple solution for the addition problem (defined in Section 3) for the case where the new specification $spec_n$ is a conjunction of a single *leads-to* property and multiple safety properties. We note that the goal of our algorithm is simply to illustrate the feasibility of this solution. Hence, although our algorithm in this section can be modified to reduce complexity further, we have chosen to present a simple (and not so efficient) solution. In Section 8, we give an intuition as to how one can implement our algorithm using counterexamples provided by model checkers.

Given a program $p = \langle S_p, I_p, \delta_p \rangle$ and a specification $spec_n = \mathcal{B} \wedge \mathcal{L}$, where $\mathcal{B}$ represents the conjunction of a set of safety properties and $\mathcal{L}$ is a $R \mapsto T$ property for state predicates R and T. Our goal is to generate a new program p' that satisfies $spec_n$ and preserves the existing specification. To guarantee that p' satisfies $\mathcal{B}$ (i.e., p' never executes a transition in the set of bad transitions $\mathcal{B}$), we exclude all transitions of p that belong to $\mathcal{B}$ (see Step 1 in Figure 1). To add the *leads-to* property $\mathcal{L} \equiv (R \mapsto T)$ to p, we need to guarantee that any computation of p' that reaches a state in R will eventually reach a state in T. Towards this end, we rank all states s based on the length of the shortest computation prefix of p from s to a state in T. In such ranking, if no state of T is reachable from s then the rank of s will be *infinity*. Also, the rank of states in T is zero.

There exist two obstacles in guaranteeing the reachability from R to T: (1) the deadlock states reachable from R, and (2) cycles reachable from R where the computations of p' may be trapped forever. We may create deadlock states by (i) removing safety-violating transitions (Step 1 in Figure 1), and (ii) making infinity-ranked states unreachable in Step 3.

To deal with the deadlock states, we make them unreachable by removing transitions that reach a deadlock state (Step 4 in Figure 1). Such removal of transitions may introduce new deadlock states that are removed in the *while* loop in Step 4. If the removal of deadlock states culminates in making an initial state deadlocked then $(R \mapsto T)$ cannot be added to p. Otherwise, we again rank all states (in Step 5) since we might have removed some deadlock states in T, and as a result, we might have created new infinity-ranked states. We repeat the above steps until no reachable state in R has the rank infinity. At this point (end of repeat-until in Step 6), there is a path from each state in R to a T state.

```
Add_UNITY(I_p: state predicate, p: set of transitions, R,T: state predicate, B: safety specification )
{ // S_p is the state space of p.

    p1 := p − {(s0, s1) | (s0, s1) ∈ B};                                                          (1)
    ∀s : s ∈ S_p : Rank(s) = the length of the shortest computation prefix of p1                  (2)
                              that starts from s and ends in a state in T;
                             //Rank(s) = ∞ means T is not reachable from s.
    repeat{
        p1 := p1 − {(s0, s1) | (s1 ∈ R) ∧ Rank(s1) = ∞};                                          (3)
        while (∃s0 :: (∀s1 : s1 ∈ S_p : (s0, s1) ∉ p1)) {                                         (4)
                    If (s0 ∉ I_p) then   p1 := p1 − {(s, s0) | (s, s0) ∈ p1};
                    else                 declare that the addition is not possible; exit();
                  }
        ∀s : s ∈ S_p : Rank(s) = the length of the shortest computation prefix of p1              (5)
                                  that starts from s and ends in a state in T;
    } until (∀s : (s ∈ R) ∧ (s is reachable in p1) : Rank(s) ≠ ∞)                                (6)
    return p1 − {(s0, s1) | Rank(s0) < Rank(s1)};                                                 (7)
}
```

Fig. 1. Adding one *leads-to* and multiple safety properties.

However, there may be cycles that are reachable from a state in R without reaching T.

To deal with such cycles from R, we remove transitions from low-ranked states to high-ranked states (Step 7 in Figure 1). In particular, if $Rank(s_0) < Rank(s_1)$ then that means there exists a shorter computation prefix from s_0 to T with respect to the computation prefix from s_1 to T. Thus, removing (s_0, s_1) will not make s_0 deadlocked. (Note that in Step 7, transitions of the form (s_0, s_1), where $Rank(s_0) = \infty$ and $Rank(s_1) = \infty$, are not removed. Hence, computations in which neither predicates R and T are reached will not be affected.)

Theorem 4.1 The algorithm Add_UNITY is sound.

Proof. Since Add_UNITY does not add any new states to S_p, we have $S_{p'} = S_p$. Likewise, Add_UNITY does not remove (respectively, introduce) any initial states; we have $I_{p'} = I_p$. The algorithm Add_UNITY only updates δ_p by excluding some transitions from δ_p in Steps 1, 3, 4, and 7. It follows that $\delta_{p'} \subseteq \delta_p$. By construction, Add_UNITY removes all deadlock states in Step 4. Thus, if Add_UNITY generates a program p' in Step 7 then reachability from R to T is guaranteed in p'. Thus, p' meets all the requirements of the addition problem. □

Theorem 4.2 The algorithm Add_UNITY is complete.

Proof. Note that any transition removed in Add_UNITY (in Steps 1, 3, and 4) must be removed in any program that meets the requirements of the addition problem. Hence, when failure is declared (in Step 4), it follows that a solution to the addition problem does not exist. □

Theorem 4.3 The time complexity of Add_UNITY algorithm is polynomial in S_p.

Proof. The proof follows from the polynomial-time complexity of each step of Add_UNITY. □

In Section 6, we demonstrate our algorithm in the local redesign of a token passing mutual exclusion program. We have also used our algorithm in the local redesign of a readers-writers program in [3].

5 Adding Two *Leads-to* Properties

In this section, we show that the addition of a UNITY specification, which is the conjunction of two *leads-to* properties, to a program is NP-complete. We show this by presenting a reduction from the 3-SAT problem to an instance of the decision problem defined below. The instance and the decision problem for adding two *leads-to* properties are as follows:

Instance. An instance of the addition problem for two *leads-to* properties consists of a program p , its state space S_p , set of initial states I_p , transitions δ_p , and $spec_n = \mathcal{L}_1 \ \wedge \ \mathcal{L}_2$, where $\mathcal{L}_1 \equiv P \mapsto Q$ and $\mathcal{L}_2 \equiv R \mapsto T$, and P, Q, R, and T are state predicates.

The decision problem.
Given is an instance of the addition problem for two *leads-to* properties:

> *Does there exist a program* p'*, its state space* $S_{p'}$*, and its set of initial states* $I_{p'}$ *such that*
> $S_p = S_{p'}$, $I_{p'} = I_p$, $\delta_{p'} \subseteq \delta_p$ *, and* p' *satisfies* $spec_n = \mathcal{L}_1 \ \wedge \ \mathcal{L}_2$*?*

The 3-SAT problem is as follows: Let $x_1, x_2, ..., x_n$ be propositional variables. Given is a Boolean formula $y = y_1 \ \wedge \ y_2 \ \cdots \ \wedge \ y_M$, where each y_j $(1 \leq j \leq M)$ is a disjunction of exactly three literals. Does there exist an assignment of truth values to $x_1, x_2, ..., x_n$ such that y is satisfiable?

Next, in Section 5.1, we present a polynomial-time mapping from 3-SAT to an instance of the decision problem. Then, in Section 5.2, we show that the 3-SAT problem is satisfiable iff the answer to the above decision problem is affirmative for the instance introduced in Section 5.1.

5.1 Mapping 3-SAT to the Addition of Two *Leads-to* Properties

We now present the mapping of an instance of the 3-SAT problem to an instance of the problem of adding two *leads-to* properties. First, we introduce the state space and the initial states of the instance of the addition problem corresponding to each variable x_i and each disjunction y_j. We also introduce the state predicates P, Q, R, and T that define $spec_n$. Then, we present the transitions of the instance corresponding to each variable x_i and each disjunction y_j.

The state space, initial states, and state predicates P, Q, R, and T. Corresponding to each variable x_i of the given 3-SAT instance, we introduce six states P_i, a_i, Q_i, R_i, b_i, and T_i, where $1 \leq i \leq n$ (see Figure 2). For each disjunction y_j, we introduce a state c_j, where $1 \leq j \leq M$, in the state space. Thus,

- $S_p = \{P_i, a_i, Q_i, R_i, b_i, T_i \mid 1 \leq i \leq n\} \cup \{c_j \mid 1 \leq j \leq M\}$
- $I_p = \{c_j \mid 1 \leq j \leq M\}$
- $P = \{P_i \mid 1 \leq i \leq n\}$, $Q = \{Q_i \mid 1 \leq i \leq n\}$, $R = \{R_i \mid 1 \leq i \leq n\}$, and $T = \{T_i \mid 1 \leq i \leq n\}$

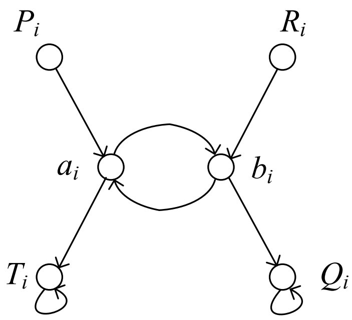

Fig. 2. Mapping of variables in the 3-SAT problem.

The program transitions. Corresponding to each variable x_i, we include transitions $(P_i, a_i), (a_i, b_i)$, (b_i, Q_i), (Q_i, Q_i), $(R_i, b_i), (b_i, a_i)$, (a_i, T_i), and (T_i, T_i) in the set of program transitions δ_p (see Figure 2). Moreover, corresponding to each disjunction y_j, we include the following transitions:

- If x_i is a literal in y_j then we include the transition (c_j, P_i).
- If $\neg x_i$ is a literal in y_j then we include the transition (c_j, R_i).

5.2 Reduction from the 3-SAT Problem

In this subsection, we show that the given instance of 3-SAT is satisfiable iff both *leads-to* properties $\mathcal{L}_1 \equiv (P \mapsto Q)$ and $\mathcal{L}_2 \equiv (R \mapsto T)$ can be added to the problem instance identified in Subsection 5.1.

Part I. First, we show that if the given instance of the 3-SAT formula is satisfiable then there exists a solution that meets the requirements of the decision problem. Since the 3-SAT formula is satisfiable, there exists an assignment of truth values to variables x_i, $1 \leq i \leq n$, so that each y_j, $1 \leq j \leq M$, is *true*. Now, we identify a program p', that is obtained by adding the *leads-to* properties $\mathcal{L}_1$ and $\mathcal{L}_2$ to program p as follows.

- The state space of p' consists of all the states of p, i.e., $S_{p'} = S_p$.
- The initial states of p' consists of all the initial states of p, i.e., $I_{p'} = I_p$.
- For each variable x_i, if x_i is *true* then we include the transitions (P_i, a_i), (a_i, b_i), (b_i, Q_i), and (Q_i, Q_i).
- For each variable x_i, if x_i is *false* then we include the transitions (R_i, b_i), (b_i, a_i), (a_i, T_i), and (T_i, T_i).
- For each disjunction y_j that contains x_i, we include the transition (c_j, P_i) if x_i is *true*.
- For each disjunction y_j that contains $\neg x_i$, we include the transition (c_j, R_i) if x_i is *false*.

As an illustration, we show the partial structure of p', for the formula $[(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)]$, where x_1 = *true*, x_2 = *false*, x_3 = *false*, and x_4 = *false* in Figure 3.

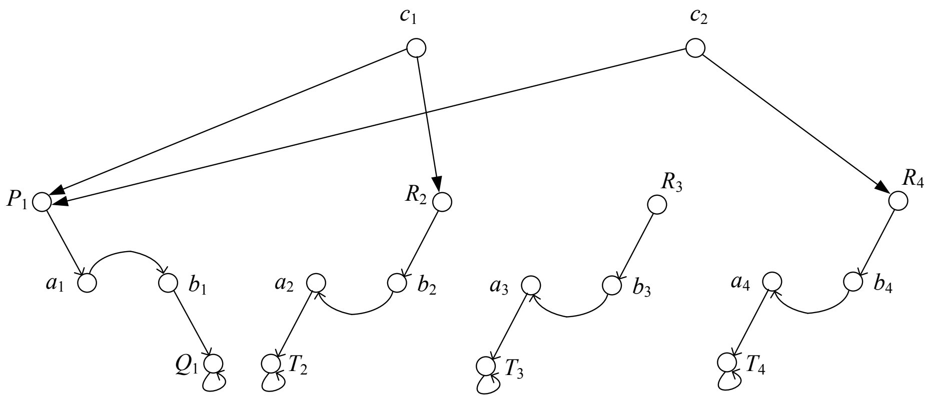

Fig. 3. The partial structure of the revised program.

Now, we show that p' meets the requirements of the decision problem.

- The first three constraints of the decision problem are trivially satisfied.
- It is easy to observe that by construction, there are no deadlock states. Hence, for the UNITY specification $spec_e$ if p satisfies $spec_e$ then p' also satisfies $spec_e$. Moreover, if a computation of p' reaches P_i from some initial state (i.e., x_i is *true*) then that computation will eventually reach Q_i and will stay there, since p' does not include the transition (b_i, a_i). Likewise, if a computation of p' reaches R_i from some initial state (i.e., x_i is *false*) then that computation will eventually reach T_i and will stay there, since p' does not include the transition (a_i, b_i). Thus, p' satisfies both $\mathcal{L}_1$ and $\mathcal{L}_2$.

Part II. Next, we show that if there exists a solution to the instance identified in Subsection 5.1, then the given 3-SAT formula is satisfiable. Let p' be the program that is obtained by adding the two *leads-to* properties to program p. Now, to obtain the solution for 3-SAT, we proceed as follows. If there exists a computation of p' where state P_i is reachable then we assign x_i the truth value *true*. Otherwise, we assign it the truth value *false*.

We now show that the above truth assignment satisfies all disjunctions. Let y_j be any disjunction and let c_j be the corresponding state in p'. Since c_j is an initial state and p' cannot deadlock, there must be some transition from c_j. This transition terminates in either P_i or R_i, for some i. If the transition from c_j terminates in P_i then y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. If the transition from c_j terminates in R_i then P_i should not be reachable. Otherwise, (i) transitions $(R_i, b_i), (b_i, a_i)$, and (a_i, T_i) must be included to ensure that $R \mapsto T$ is satisfied, and (ii) transitions $(P_i, a_i), (a_i, b_i)$, and (b_i, Q_i) must also be included to guarantee that $P \mapsto Q$ is satisfied. Since the inclusion of all six transitions (P_i, a_i), (a_i, b_i), (b_i, Q_i), (R_i, b_i), (b_i, a_i), and (a_i, T_i) causes violation of $P \mapsto Q$ and $R \mapsto T$, it follows that P_i must not be reached in any computation of p' if R_i is reachable. Thus, if

R_i is reachable then x_i will be assigned the truth value *false*. Since in this case y_j contains $\neg x_i$, the disjunction y_j evaluates to *true*. Therefore, the assignment of values considered above is a satisfying truth assignment for the given 3-SAT formula. □

Theorem 5.1 The addition of two *leads-to* properties to UNITY programs is NP-complete.

Proof. The NP-hardness of adding two *leads-to* properties follows from the reduction presented in this section. Also, given a solution (in terms of p' consisting of $S_{p'}, I_{p'}, \delta_{p'}$) to the instance of the decision problem, one can verify the requirements (1) $S_{p'} = S_p$, (2) $I_{p'} = I_p$, (3) $\delta_{p'} \subseteq \delta_p$, and (4) p' satisfies $spec_n$ in polynomial time. Thus, the membership to NP follows. Therefore, the problem of adding two *leads-to* properties is NP-complete. □

6 Example: Mutual Exclusion

In this section, we illustrate the role of the algorithm Add_UNITY in deciding about local or comprehensive redesign of a token passing mutual exclusion (ME) program. We use Dijkstra's guarded commands (*actions*) [4] as the 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 $\}$.

The initial ME program has two competing processes P_1 and P_2. Each process P_j $(j = 0, 1)$ has three Boolean variables n_j, c_j, and t_j, where (i) t_j represents whether or not P_j is trying to enter its critical section (i.e., trying section), (ii) c_j represents whether or not P_j is in its critical section, and (iii) n_j represents whether or not P_j intends to enter its trying section (i.e., non-trying section). The variables of P_j are mututally exclusive; i.e., the condition $(t_j \Rightarrow (\neg n_j \wedge \neg c_j)) \wedge (n_j \Rightarrow (\neg t_j \wedge \neg c_j)) \wedge (c_j \Rightarrow (\neg n_j \wedge \neg t_j))$ holds. We denote a state of ME by $\langle s_0, s_1 \rangle$, where s_0 represents the state of P_0 and s_1 represents the state of P_1. Also, we represent the actions of a process j $(j = 0, 1\ )$ as follows:

$$
\begin{array}{llcl}
ME1_j: & n_j & \longrightarrow & t_j := true; n_j := false; \\
ME2_j: & t_j & \longrightarrow & c_j := true; t_j := false; \\
ME3_j: & c_j & \longrightarrow & n_j := true; c_j := false;
\end{array}
$$

For simplicity, we illustrate the reachability graph of the initial ME program in Figure 4 that shows all reachable states from the initial state s_{init} , where both processes are in there non-critical sections. We have annotated each transition with the index of the process that executes that transition.

In the initial state of ME, both processes are in their non-trying section (i.e., $n_0 = true$ and $n_1 = true$). The ME program satisfies its safety property that stipulates P_0 and P_1 must not enter the critical section simultaneously (i.e., $Invariant(\neg(c_0 \wedge c_1))$). Also, the initial ME program only satisfies $c_j \mapsto n_j$. Next, we trace Add_UNITY to add the leads-to property $t_0 \mapsto c_0$ to ME while preserving $c_0 \mapsto n_0$. For reasons of space, we omit the addition of $t_1 \mapsto c_1$ as it is similar to the addition of $t_0 \mapsto c_0$.

Step 1. Since ME already satisfies its safety property, no transitions are removed at the first step of Add_UNITY.

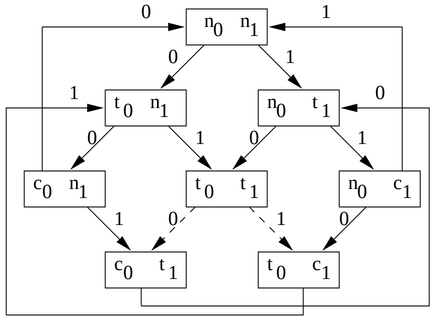

Fig. 4. The reachability graph of program ME.

Step 2. The Add_UNITY algorithm ranks all states based on their shortest computation prefix to states where c_0 is *true*. As a result, the rank of $\langle t_0, t_1\rangle$ becomes 1 and the rank of $\langle t_0, c_1\rangle$ becomes 2.
Step 3. Since there exist no states with rank ∞, Add_UNITY does not remove any transitions in Step 3.
Step 4. Since the execution of Steps 2 and 3 does not create any deadlocked states, Add_UNITY does not modify the program structure in Step 4.
Step 5 and 6. The ranking of the states will not be changed in Step 5. Also, Add_UNITY exits the repeat-until loop since no state where t_0 holds has a rank of ∞.
Step 7. Finally, in Step 7, Add_UNITY removes the transition $\langle t_0, t_1\rangle \rightarrow \langle t_0, c_1\rangle$ since the rank of $\langle t_0, t_1\rangle$ is 1 and the rank of $\langle t_0, c_1\rangle$ is 2.

A similar execution of Add_UNITY for the addition of $t_1 \mapsto c_1$ results in the synthesis of the following ($\oplus$ denotes modulo 2 addition):

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

Note that, the above program does not satisfy $n_j \mapsto t_j$. Now, if we use Add_UNITY for the addition of $n_j \mapsto t_j$, while preserving $t_j \mapsto c_j$ and $c_j \mapsto n_j$, then Add_UNITY will declare failure because the initial state will be deadlocked. Based on the completeness of Add_UNITY, it follows that the initial program cannot be revised to a program that simultaneously satisfies the above *leads-to* properties. This is an interesting result that enlightens designers to search for other solutions where one adds new variables and computations to the ME program (e.g., Peterson's solution) instead of spending time on modifying the initial ME program.

7 Related Work

In this section, we illustrate how the contributions of this paper differ from existing approaches for program synthesis and verification. Existing synthesis methods in the literature mostly focus on deriving the synchronization skeleton of a program from its specification (expressed in terms of temporal logic expressions or finite-state automata) [5–11], where the synchronization skeleton of a program is an *abstract structure* of the code of the program implementing inter-process synchronization. Although such synthesis methods may have 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 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 algorithm (cf. Figure 1) is the set of transitions of a program, our approach has the potential to reuse those transitions in incremental synthesis of a revised version of the input program.

The algorithms for automatic addition of fault-tolerance [12–16] add fault-tolerance concerns to existing programs in the presence of faults, and guarantee not to add new behaviors to that program in the absence of faults. The problem of adding fault-tolerance is orthogonal to the problem of adding UNITY properties in that one could use the algorithms of [12–16] to add fault-tolerance concerns to a UNITY program synthesized by the algorithm presented in this paper. On the other hand, we plan to investigate the addition of UNITY properties to fault-tolerant programs while preserving their fault-tolerance properties.

Run-time verification. Runtime verification techniques focus on monitoring the program behavior at runtime with respect to a given specification [17]. Also, such techniques provide a mechanism for ensuring the correctness of program execution after monitoring violations of desired properties [18]. Such approaches mostly focus on the verification of safety properties [19–22] and also provide mechanisms for exception handling and dealing with deadlocks at runtime. By contrast, our focus is on off-line addition of UNITY properties to programs where we ensure that the synthesized program satisfies its existing and newly added properties. Also, to the best of our knowledge, the runtime verification of *leads-to* properties is still an open question.

8 Discussion

In this section, we address some questions raised about the limitations and the applications of the results presented in this paper. We proceed as follows:

Stepwise application of Add_UNITY. The algorithm Add_UNITY can be combined in a stepwise fashion. While such stepwise combination of Add_UNITY to add multiple *leads-to* properties will be sound, it is not complete. This is due to the fact that during the addition of the first *leads-to* property, the transitions removed in the last step (Step 7 in Figure 1) may cause failure in adding the subsequent *leads-to* property. Therefore, this does not contradict the NP-completeness result in Section 5.

Addition of other UNITY properties. The algorithm Add_UNITY shows that it is possible to add several safety (*stable, invariant* and *unless*) properties and one *leads-to* property in polynomial time. Since *ensures* is a conjunction of *unless* and *leads-to* properties, this algorithm can be trivially extended to deal with the case where one adds several safety properties and an *ensures* property. Also, one can use Add_UNITY to add the *until* property in Linear Temporal Logic (LTL) [23] to programs as *ensures* is semantically the same as *until* in LTL.

However, in the context of adding multiple *leads-to* properties, there are several open questions. For example, is it possible to combine these *leads-to* properties with other (specific) properties to obtain efficient solutions? To illustrate this, it is straightforward to observe that adding '$invariant(\neg P) \wedge (P \mapsto Q) \wedge (R \mapsto T)$' can be added efficiently, as it corresponds to adding '$invariant(\neg P) \wedge (R \mapsto T)$'. Moreover, the complexity of adding two *ensures* properties is still an open question. (Note that the complexity of adding two *ensures* properties does not necessarily follow from the results in Section 5; as discussed earlier in this paragraph, combining *leads-to* properties with certain safety properties, does permit polynomial time solutions.)

Implementing Add_UNITY using model checking. The algorithm Add_UNITY can also be implemented with the help of a model checker as follows: For this exposition, consider the case where a program, say p, is specified as a set of transitions, as defined in Section 2. When p is checked with the model checker with respect to a *leads-to* property $(R \mapsto T)$ and found to be incorrect, the counterexamples will be of one of the following two forms: (1) There exists a state s_d such that s_d is reachable in computations of p and s_d is a deadlocked state, or (2) There exists a state, say $s_r \in R$ that is reachable in a program computation and that program computation can be extended to reach a cycle, say $s_0, s_1, \cdots, s_n (= s_0)$ such that T is never satisfied. In the former case, transitions terminating in s_d need to be removed. In the latter case, we need to check if there exists a computation prefix of p that starts in one of the states in the cycle and reaches T. (This case could also be checked with a model checker.) If such a computation prefix does not exist then the state s_r and all its incident transitions should be removed. If such a computation prefix exists and s_j is the last state from the cycle to appear on that path then the transition (s_j, s_{j+1}) in the cycle should be removed. After removing the transitions in this fashion, we can repeat the process with the new program until a solution is found. (We leave it to the reader to verify that this approach is also sound and complete.)

The choice of the initial program. The algorithm Add_UNITY takes the initial program p and adds a set of UNITY properties to p if possible. Thus, as far as the properties of the Add_UNITY 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 maximal non-determinism, then the chance of a successful addition is higher. This issue is particularly important for a step-wise application of the Add_UNITY algorithm.

9 Conclusion and Future Work

In this paper, we focused on the problem of revising UNITY [1] programs where one adds a conjunction of UNITY properties *unless, stable, invariant, ensures*, and *leads-to* to an existing program to provide new functionalities while preserving the existing functionalities. This is an important problem given the dynamic nature of the requirements of computing systems, where developers need to constantly revise existing programs due to newly-discovered user requirements. In particular, we formally defined the problem of adding UNITY properties to programs. Afterwards, we presented a sound and complete algorithm for such addition where one automatically (i) verifies if it is possible to add a conjunction of UNITY properties to a program and preserve the existing properties, and (ii) adds a conjunction of UNITY properties to a program if such addition is possible.

More importantly, we showed that if one adds a single *leads-to* property and a conjunction of *unless, stable*, and *invariant* properties to a program then the complexity of such addition will be polynomial in program state space. However, in general, we showed a surprising result that simultaneous addition of more than one *leads-to* properties to a program is NP-complete. Hence, revising UNITY programs would be significantly easier if one added a single *leads-to* property instead of adding two *leads-to* properties. Since *ensures* can be expressed as the conjunction of an *unless* property and a *leads-to* property, the algorithm presented in this paper for adding a *leads-to* property and a conjunction of *unless, stable*, and *invariant* properties can be used for the addition of *ensures* property as well. Nonetheless, to the best of our knowledge, the complexity of adding two *ensures* properties to UNITY programs is still an open problem.

To extend the results of this paper, we plan to integrate the algorithm presented in this paper with model checking algorithms to provide automated assistance for developers. As a result, if the model checking of a model with respect to a UNITY property fails then our algorithm automatically (i) determines whether or not the model is *fixable*, and (ii) fixes the model if it is fixable.

References

1. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. Ali Ebnenasir and Sandeep Kulkarni. Automatic addition of liveness. Technical Report MSU-CSE-04-22, Department of Computer Science, Michigan State University, East Lansing, Michigan, June 2004.
4. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1990.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.

10. P. Attie. Synthesis of large concurrent programs via pairwise composition. *CONCUR'99: 10th International Conference on Concurrency Theory*, 1999.
11. 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.
12. 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*, pages 82 – 93, 2000.
13. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. *Symposium on Reliable Distributed Systems*, pages 130–139, 2001.
14. S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 337–344, 2002.
15. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 441–449, 2003.
16. S. S. Kulkarni and Ali Ebnenasir. Automated synthesis of multitolerance. *In Proceedings of the International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy*, pages 209 – 218, June 28 - July 1 2004.
17. K. Havelund and G. Rosu. Runtime verification. *Formal Methods in System Design. Special issue dedicated to RV'01*, 24(2), 2004.
18. F. Chen, M. D'Amorim, and G. Rosu. A formal monitoring-based framework for software development and analysis. *Sixth International Conference on Formal Engineering Methods (ICFEM)*, pages 357–372, November 2004.
19. Bernd Fisher, Johann Schumann, and Mike Whalen. Synthesizing certified code. *In Proceedings Formal Methods Europe(FME'02). Copenhagen, Denmark. LNAI, Springer*, 2002.
20. 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.
21. 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.
22. 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.
23. E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.