Complexity Results in Revising UNITY Programs

BORZOO BONAKDARPOUR
Michigan State University
ALI EBNENASIR
Michigan Technological University
SANDEEP S. KULKARNI
Michigan State University

We concentrate on automatic revision of untimed and real-time programs with respect to UNITY properties. The main focus of this paper is to identify instances where addition of UNITY properties can be achieved efficiently (in polynomial-time) and where the problem of adding UNITY properties is difficult (NP-complete). Regarding efficient revision, we present a sound and complete algorithm that adds a single *leads-to* property (respectively, *bounded-time leads-to* property) and a conjunction of *unless, stable*, and *invariant* properties (respectively, *bounded-time unless* and *stable*) to an existing untimed (respectively, real-time) UNITY program in polynomial-time in the state space (respectively, region graph) of the given program. Regarding hardness results, we show that (1) while one *leads-to* (respectively, *ensures*) property can be added in polynomial-time, the problem of adding two such properties (or any combination of *leads-to* and *ensures*) is NP-complete, (2) if maximum non-determinism is desired then the problem of adding even a single *leads-to* property is NP-complete, and (3) the problem of providing maximum non-determinism while adding a single *bounded-time leads-to* property to a real-time program is NP-complete (in the size of the program's region graph) even if the original program satisfies the corresponding *unbounded leads-to* property.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying; Verifying and Reasoning about Programs; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures; D.1.2 [**Programming Techniques**]: Automatic Programming—*Program modification*; *synthesis*; *transformation*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time and embedded systems*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: UNITY, Formal Methods

1. INTRODUCTION

In this paper, we focus on automated revision of untimed and real-time existing programs where all variables can be read and written in one atomic step with respect

Addresses: B. Bonakdarpour and S. Kulkarni (`{borzoo,sandeep}@cse.msu.edu`) 3115 Engineering Building, Michigan State University, East Lansing, MI 48824; A. Ebnenasir (`aebnenas@mtu.edu`) 221 Rekhi Hall, Michigan Technological University, Houghton MI 49931. This work was partially sponsored by NSF CAREER CCR-0092724 and a grant from Michigan Technological University.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

to UNITY properties [Chandy and Misra 1988; Carruth 1994]. To motivate the application of this work, consider the following scenario: *a designer checks whether a program satisfies a property using a model checker. The model checker provides a counterexample demonstrating that the program does not satisfy the property.* The property in the scenario can be a known property that a designer is dealing with, or, it may be a newly identified property due to an incomplete specification or modifications in the specification. In this scenario, the designer needs to manipulate the given model so that it satisfies the property (while ensuring that the remaining properties continue to be satisfied).

There exist two automated ways in which one can deal with the above scenario: (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 develops a new program from scratch (possibly with new behaviors) that implements the new property while preserving existing properties. Several approaches exist [Emerson and Clarke 1982; Manna and Wolper 1984; Attie and Emerson 2001; Ramadge and Wonham 1989; Lin and Wonham 1990; Lafortune and Lin 1992; Rudie et al. 2003; Rohloff 2004; Wallmeier et al. 2003; Thomas 2002; Maler et al. 2006; Alur et al. 1996] for comprehensive redesign most of which require a complete specification and have less emphasis on *reuse* in the face of change in program specifications. While these approaches play an important role in comprehensive redesign of program models from formal specifications, we believe that in the face of increasingly dynamic systems with evolving requirements, local redesign is highly desirable. This is due to the fact that local redesign (1) has the potential to reuse certain computations of an existing program without actually exacerbating the complexity of the state space explosion problem (unlike approaches based on automata-theoretic product calculation), and (2) can be applied in cases where complete system specification is unavailable.

We expect that an algorithm for local redesign would be especially useful if it were *simultaneously* sound and complete. A sound algorithm ensures that a redesigned program meets a property of interest (in addition to preserving existing specification), i.e., the redesigned program is correct-by-construction. A complete algorithm provides an insight for designers to decide whether a program can be redesigned locally or it should be redesigned from scratch to satisfy the property while preserving its exiting properties. Such automated assistance to a 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*.

Another application of local redesign is in synthesizing recovery paths in fault-tolerant systems. When a program is subject to a set of faults, it may reach a state that is not legitimate. Thus, safe recovery in a finite number of steps is needed to ensure that the program reaches a legitimate state. One way to achieve such recovery is to first augment the program with all possible recovery transitions and then revise the program with respect to a safety property to ensure that recovery is safe and a progress property to ensure that a legitimate state is eventually reached.

With this motivation, we present an incremental method for adding both untimed and real-time UNITY properties to programs. 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 [Chandy and Misra 1988; Carruth 1994]. We expect to benefit from simplicity and generality of UNITY in automatic design of programs.

Contributions. The basic UNITY properties from [Chandy and Misra 1988] are *safety* properties *unless, stable* and, *invariant* where *something bad will never happen*, and *progress* properties *ensures* and *leads-to* where *something good will eventually happen*. In the context of real-time, Carruth extends these properties such that the occurrence of the good and bad things are considered in a *bounded* amount of time [Carruth 1994]. We note that since self-stabilization [Dijkstra 1974] properties can be expressed as a conjunction of *stable* and *leads-to* properties, the results in this paper can be trivially used to achieve automated addition of self-stabilization properties to existing programs as well. The main results of this paper are as follows:

—We formally define the problem of adding UNITY properties to programs.

—We introduce a polynomial-time sound and complete algorithm for simultaneous addition of a single *leads-to* property and a conjunction of safety properties to untimed programs. As a result, since *ensures* can be expressed in terms of a *leads-to* and *unless* properties, our algorithm is able to add a single *ensures* property and a conjunction of safety properties as well.

—We present a counterintuitive result where we show that addition of two or more progress (i.e., *ensures* and *leads-to*) properties to an untimed program is NP-complete. Based on this result, we find that adding one progress property is significantly easier than addition of multiple such properties.

—We show that the problem of adding a single *leads-to* property to an untimed program while preserving maximum non-determinism is NP-complete.

—We propose a sound and complete polynomial-time algorithm in the size of the input program's region graph (a time-abstract bisimulation representation of real-time programs) for simultaneous addition of a single *bounded-time leads-to* property along with a conjunction of safety properties. Similar to the untimed case, our algorithm is able to add a single *bounded-time ensures* property and a conjunction of safety properties as well.

—We show another counterintuitive result that the problem of providing maximum non-determinism while adding a single *bounded-time leads-to* property to a real-time program is NP-complete (in the size of the input program's region graph) even if the original program satisfies the corresponding *unbounded leads-to* property.

Organization of the paper. The rest of the paper is organized as follows. We present our polynomial-time sound and complete algorithm and NP-completeness results on automated addition of untimed UNITY properties in Sections 2 and 3, respectively. We present our sound and complete algorithm for adding real-time UNITY properties and NP-completeness result in Sections 4 and 5, respectively. Then, in Section 6, we compare the results of this paper with the related work. Finally, we make concluding remarks in Section 7.

2. REVISING UNTIMED UNITY PROGRAMS: POLYNOMIAL-TIME ALGORITHM

In this section, we focus on automated revision of *untimed* UNITY programs. First, we present the preliminary concepts.

Untimed 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 infinite 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) $\forall j > 0 : (s_{j-1}, s_j) \in \delta_p$ holds, and (3) if σ reaches a *terminating* state s_f where there does not exist s such that $s \neq s_f$ and $(s_f, s) \in \delta_p$ then we extend σ to an infinite computation by stuttering at s_f using transition (s_f, s_f). If a computation σ reaches a terminating state s_d such that there is no outgoing transition (or a self-loop) from s_d, then s_d is a *deadlock state* and σ is a *deadlocked computation*. A sequence of states $\langle s_0, s_1 \cdots s_n \rangle$ is a *computation prefix* of p iff $\forall j \mid 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$.

Untimed UNITY properties [Chandy and Misra 1988]. Let P and Q be two arbitrary state predicates.

—(*Unless*) An infinite sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfies P *unless* Q iff $\forall i \geq 0 : (s_i \in (P \cap \neg Q)) \Rightarrow (s_{i+1} \in (P \cup Q))$. Intuitively, if P holds in any 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 $\sigma = \langle s_0, s_1 \cdots \rangle$ 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 $\sigma = \langle s_0, s_1 \cdots \rangle$ satisfies *invariant*(P) iff $s_0 \in P$ and σ satisfies *stable*(P). An invariant property always holds.

—(*Ensures*) An infinite sequence of states $\sigma = \langle s_0, s_1 \cdots \rangle$ satisfies P *ensures* Q iff (σ satisfies P *unless* Q) and $(\forall i \geq 0 : (s_i \in P) \Rightarrow (\exists j \geq i : s_j \in Q))$. In other words, if P becomes true in s_i, there exists a state s_j where Q eventually becomes true in s_j and P remains true everywhere between s_i and s_j.

—(*Leads-to* $\mapsto$) An infinite sequence of states $\sigma = \langle s_0, s_1 \cdots \rangle$ satisfies $P \mapsto Q$ iff $(\forall i \geq 0 : (s_i \in P) \Rightarrow (\exists j \geq i : s_j \in Q))$. In other words, if P holds in s_i then there exists a state s_j where Q eventually holds and $i \leq j$.

We define a *specification SPEC* as a conjunction of the above UNITY properties (i.e., $SPEC \equiv \mathcal{L}_1 \wedge \mathcal{L}_2 \cdots \mathcal{L}_n$). A sequence of states $\sigma = \langle s_0, s_1 \cdots \rangle$ *satisfies SPEC* iff $(\forall i \mid 1 \leq i \leq n : \sigma$ satisfies $\mathcal{L}_i)$. We say that program p *satisfies* a UNITY specification, $SPEC$, iff all computations of p satisfy $SPEC$.

The properties *unless, stable*, and *invariant* are called *safety* properties and the properties *ensures* and *leads-to* are called *progress* (or *liveness*) properties. The safety UNITY 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 safety UNITY properties, we assume that they are represented as a set of transitions $\mathcal{B} \subseteq S_p \times S_p$ that should not occur in any computation.

Differences with standard UNITY. In our formal framework, unlike standard UNITY in which *interleaved fairness* is assumed, we assume that all program computations are unfair. This assumption is necessary in dealing with polynomial-time addition of UNITY progress properties to programs. We also note that the definition of *ensures* property is slightly different from that in [Chandy and Misra 1988]. Precisely, in Chandy and Misra's definition, (P *ensures* Q) implies that (1) P *leads-to* Q, (2) P *unless* Q, and (3) there is at least one action that always establishes Q whenever it is executed in a state where P is true and Q is false. Since, we do not model actions explicitly in our work, we have removed the third requirement. Finally, as described earlier, in this paper, in the context of untimed programs, our focus is only on programs whose *finite* set of discrete variables is of *finite* domain.

2.1 Problem Statement

Given a program $p = \langle S_p, I_p, \delta_p \rangle$ and a UNITY specification $SPEC_n$, our goal is to revise p so that the revised program (denoted $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$) satisfies $SPEC_n$ while preserving its existing UNITY specification $SPEC_e$. We consider the case where specification $SPEC_e$ may be unknown. Thus, during the revision, 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 $\delta_{p'}$. Observe that if $S_{p'}$ contains states that are not in S_p, there is no guarantee that the correctness of p with respect to $SPEC_e$ can be reused to ensure p' satisfies $SPEC_e$. Also, 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 revision. Finally, we require that $\delta_{p'}$ should be a subset of δ_p. Note that not all transitions of δ_p may be preserved in p'. Hence, we must ensure that p' does not deadlock. Based on the definition of UNITY specifications, if (i) $\delta_{p'} \subseteq \delta_p$, (ii) p' does not deadlock, and (iii) p satisfies $SPEC_e$, then p' also satisfies $SPEC_e$. Thus, the problem statement is defined as follows:

Problem Statement 2.1 Given a program $p = \langle S_p, I_p, \delta_p \rangle$ and a UNITY specification $SPEC_n$, identify $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ such that:

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

Note that the requirement of deadlock freedom is not explicitly specified in the above problem statement, as it follows from 'p' satisfies $SPEC_n$'. Throughout the paper, we use 'revision of p' with respect to a property $\mathcal{L}$ and 'addition of $\mathcal{L}$' to p interchangeably.

2.2 Adding a Single Leads-to and Multiple Safety Properties

In this section, we present a simple solution for Problem Statement 2.1 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 a polynomial-time solution. Hence, although our algorithm in this section can be modified to reduce the complexity further, we have chosen to present a simple (and not so efficient) solution (see Algorithm 1).

Let $p = \langle S_p, I_p, \delta_p \rangle$ be a program and specification $SPEC_n \equiv \mathcal{B} \wedge \mathcal{L}$, where $\mathcal{B}$ represents the conjunction of a set of safety properties and $\mathcal{L} \equiv (R \mapsto T)$ for state predicates R and T. In order to guarantee that the revised program p' satisfies $\mathcal{B}$ (i.e., p' never executes a transition in the set of bad transitions $\mathcal{B}$), we simply remove all transitions in $\mathcal{B}$ from p (Step 1).

In order 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 in S_p based on the length of the shortest computation prefix of p from s to a state in T (Step 2). In such a 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 of T from R: (1) deadlock states reachable from R, and (2) cycles reachable from R in which computations of p' may be trapped forever. In addition to possible existing deadlock states in p, our algorithm may also introduce deadlock states by (i) removing safety-violating transitions (Step 1), and (ii) making infinity-ranked states in R unreachable in Step 4.

Regarding deadlock states, our approach is to make them unreachable (Steps 5-12). Such removal of transitions may introduce new deadlock states that are removed in the *while* loop. 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 (Step 13) as we might have removed some deadlock states in T, and consequently, 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 the repeat-until loop), there is a path from each state in R to a state in T. However, there

Algorithm 1 Add_UNITY

Input: untime program $\langle S_p, I_p, \delta_p \rangle$, *leads-to* property $R \mapsto T$, and safety specification $\mathcal{B}$.
Output: revised program $\langle S_{p'}, I_{p'}, \delta_{p'} \rangle$.

1: $\delta_{p_1} := \delta_p - \{(s_0, s_1) \mid (s_0, s_1) \in \mathcal{B}\}$;
2: $\forall s \in S_p : \mathit{Rank}(s) =$ the length of the shortest computation prefix of δ_{p_1} that starts from s and ends in a state in T; $\triangleright$ *$\mathit{Rank}(s) = \infty$ means T is not reachable from s.*
3: **repeat**
4: $\quad \delta_{p_1} := \delta_{p_1} - \{(s_0, s_1) \mid (s_1 \in R) \wedge \mathit{Rank}(s_1) = \infty\}$;
5: $\quad$ **while** $(\exists s_0 \in S_p : (\forall s_1 \in S_p : (s_0, s_1) \notin \delta_{p_1}))$ **do**
6: $\quad\quad$ **if** $(s_0 \notin I_p)$ **then**
7: $\quad\quad\quad \delta_{p_1} := \delta_{p_1} - \{(s, s_0) \mid (s, s_0) \in \delta_{p_1}\}$;
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ `declare that the addition is not possible`;
10: $\quad\quad\quad$ **exit**();
11: $\quad\quad$ **end if**
12: $\quad$ **end while**
13: $\quad \forall s \in S_p : \mathit{Rank}(s) =$ the length of the shortest computation prefix of δ_{p_1} that starts from s and ends in a state in T;
14: **until** $(\forall s \mid (s \in R) \wedge (s$ is reachable from I_p using $\delta_{p_1}) : \mathit{Rank}(s) \neq \infty)$
15: **return** $\delta_{p_1} - \{(s_0, s_1) \mid (\mathit{Rank}(s_0) > 0) \wedge (\mathit{Rank}(s_0) \neq \infty) \wedge (\mathit{Rank}(s_0) \leq \mathit{Rank}(s_1))\}$;

may exist a computation prefix $\langle s_0, s_1, \cdots, s_n \rangle$ such that (1) $s_0 \in R$, (2) $s_n \in T$, (3) for all $i \in \{1..n-1\} : s_i \notin T$, and (4) $\exists j \in \{2..n-1\}$ where s_j is on a cycle.

To deal with such cycles, we retain transitions from high-ranked states to low-ranked states (Step 15). In particular, if $Rank(s_0) \leq Rank(s_1)$ then it means there exists a computation prefix of shorter or equal length from s_0 to T as compared to the computation prefix from s_1 to T. Thus, removing (s_0, s_1) will not make s_0 deadlocked. Notice that in Step 15, transitions of the form (s_0, s_1), where $Rank(s_0) = \infty$ and $Rank(s_1) = \infty$, are not removed. Also, we ensure that no transitions that originate from T is removed. Hence, computations in which neither predicates R and T are reached will not be affected.

Remark. We note that since *ensures* can be expressed as a conjunction of an *unless* property and a *leads-to* property, our algorithm is able to add an *ensures* property as well.

Theorem 2.2 *The* Add_UNITY *algorithm is sound and complete.*

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 Add_UNITY algorithm only updates δ_p by excluding some transitions from δ_p in Steps 1, 4, 7, and 15. It follows that $\delta_{p'} \subseteq \delta_p$. By construction, if the Add_UNITY algorithm generates a program p' in Step 15 then reachability from R to T is guaranteed in p'. Thus, p' meets all the requirements of Problem Statement 2.1.

We now show that the algorithm is complete. Note that any transition removed in Add_UNITY (in Steps 1, 4, and 7) must be removed in any program that meets the requirements of Problem Statement 2.1. Hence, if failure is declared (in Step 9), there exists no solution to Problem Statement 2.1. □

Theorem 2.3 *The complexity of* Add_UNITY *algorithm is polynomial-time in* S_p. □

Remark. We would like to note that soundness and completeness of Add_UNITY are preserved for the case where the revised program is allowed to have a subset of initial states of the original program. For such a case, the algorithm would fail only if all initial states are removed.

2.3 Example: Readers-Writers Program

In this section, we illustrate the application of the Add_UNITY algorithm in local redesign of a program for the readers-writers problem [Chandy and Misra 1988]. We use Dijkstra's guarded commands (*actions*) [Dijkstra 1990] as a shorthand for representing the set of program transitions. A guarded command $g \rightarrow st$ captures the transitions $\{(s_0, s_1) \mid$ 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 Readers-Writers (RW) program[1]. Multiple writer processes wait in an

[1] A Promela model of the revised program is available at `http://www.cs.mtu.edu/~aebnenas/research/rdrs-wrtrs.txt`. It can be easily verified by the model checker Spin [Holzmann 1997].

infinite external queue to be picked by the program. RW contains a finite internal queue of size 2 that is managed by a *queue manager* process, which selects writers from an external queue and places them in the internal queue. The selected writer has access to a shared buffer to which other processes have access as well. At any time only one writer is allowed to write the buffer. The reader processes can read the shared buffer. The program has three integer variables $0 \leq nr \leq N$, $0 \leq nw \leq N$, and $0 \leq nq \leq 2$ that are initially 0, where N denotes the total number of processes. Specifically, 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. The program contains Boolean variables rd_j $(1 \leq j \leq N)$, and wrq that respectively represent whether or not the reader R_j is reading the buffer, and at least a writer is 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* when a process is writing the buffer.

Safety specification. The safety specification, $\mathcal{B}_{RW}$, 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:

$$\mathcal{B}_{RW} = \{(s_0, s_1) \mid (nw(s_1) > 1) \ \vee \ ((nr(s_1) \neq 0) \ \wedge \ nw(s_1) \neq 0))\}$$

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 $0 \leq (N - (nr + N \cdot nw))$. For ease of presentation, we represent the expression $(N - (nr + N \cdot nw))$ with the variable K.

Actions of RW. The actions of the writer processes in the original program are as follows:

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

When there exists a process ready for writing in the internal queue (i.e., $nq > 0$) and no process is using the buffer (i.e., $K \geq 3$), 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 (see action W_1). This way, the queue manager lets other waiting writers in. When the writer finishes its writing activity in the buffer, it exits by decrementing the value of nw (see action W_2).

The following parameterized actions represent the transitions of the readers as the structures of the readers are symmetric:

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

The condition $K > 1$ holds if no writer process is writing the buffer an at most $N - 1$ readers exist. Thus, if a reader process is not already in reading status and no writer is waiting to write the buffer (see action R_{j_1}) then the reader can read the buffer. (The original program gives the priority to the writers.) When a reader process R_j completes 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.

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

Once the queue manager selects a waiting writer, it increments the value of nq and sets wrq to *true* in order to show that a writer is waiting in the internal queue. We consider a version of the RW program where we have two readers R_0 and R_1 and one writer (i.e., $N = 3$).

Initial states. Let $\langle nr, nw, nq, rd_0, rd_1, wrq\rangle$ denote the state of the RW program. We consider the initial state $\langle 0, 0, 0, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ for the RW program.

The desired *leads-to* property. The initial program satisfies the safety specification $\mathcal{B}_{RW}$, however, 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. Readers may also wait forever due to continuous presence of writers in the internal queue. Thus, the desired *leads-to* property for a reader R_j, $j \in \{0, 1\}$, is $(0 \le K) \mapsto (rd_j)$ and the program should satisfy $(nq > 0 \mapsto (nw = 1))$ to ensure that writers have progress. In this example, we present only the redesign of RW for the property $(0 \le K) \mapsto (rd_0)$ for the reader R_0. As such, in the property $R \mapsto T$ in the input of Add_UNITY, the state predicate R is equal to $0 \le K$ and the state predicate T equals to rd_0 (see input parameters of Algorithm 1).

Adding *leads-to* using Add_UNITY. We trace the execution of Add_UNITY for the addition of $(0 \le K) \mapsto (rd_0)$ to the RW program.

—*Step 1.* Since the initial program satisfies its safety specification $\mathcal{B}_{RW}$, Step 1 of the Add_UNITY algorithm would not eliminate any transitions.

—*Step 2.* Rank 0 includes eight reachable states where $rd_0 = true$. These states are as follows: $\langle 1, 0, 0, \mathit{true}, \mathit{false}, \mathit{false}\rangle$, $\langle 1, 0, 1, \mathit{true}, \mathit{false}, \mathit{true}\rangle$, $\langle 2, 0, 0, \mathit{true}, \mathit{true}, \mathit{false}\rangle$, $\langle 1, 0, 2, \mathit{true}, \mathit{false}, \mathit{true}\rangle$, $\langle 2, 0, 1, \mathit{true}, \mathit{true}, \mathit{true}\rangle$, $\langle 2, 0, 2, \mathit{true}, \mathit{true}, \mathit{true}\rangle$, $\langle 2, 0, 1, \mathit{true}, \mathit{true}, \mathit{false}\rangle$, and $\langle 1, 0, 1, \mathit{true}, \mathit{false}, \mathit{false}\rangle$. From the initial state $\langle 0, 0, 0, \mathit{false}, \mathit{false}, \mathit{false}\rangle$, the reader R_0 can read the buffer and the program reaches the state $\langle 1, 0, 0, \mathit{true}, \mathit{false}, \mathit{false}\rangle$. Thus, the rank of the initial state is 1. Moreover, the reader R_0 can read the buffer from the states $\langle 1, 0, 0, \mathit{false}, \mathit{true}, \mathit{false}\rangle$, $\langle 1, 0, 1, \mathit{false}, \mathit{true}, \mathit{false}\rangle$, and $\langle 0, 0, 1, \mathit{false}, \mathit{false}, \mathit{false}\rangle$. As a result, the program state changes to a state in Rank 0.
The states $\langle 0, 1, 0, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ and $\langle 0, 1, 1, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ have Rank 2 as the execution of action W_2 from these states changes the program state to a state in Rank 1. Likewise, the states $\langle 0, 0, 1, \mathit{false}, \mathit{false}, \mathit{true}\rangle$ and $\langle 0, 0, 2, \mathit{false}, \mathit{false}, \mathit{true}\rangle$ get Rank 3. Rank 4 includes $\langle 1, 0, 1, \mathit{false}, \mathit{true}, \mathit{true}\rangle$, $\langle 0, 1, 1, \mathit{false}, \mathit{false}, \mathit{true}\rangle$, $\langle 0, 1, 2, \mathit{false}, \mathit{false}, \mathit{true}\rangle$, and $\langle 1, 0, 2, \mathit{false}, \mathit{true}, \mathit{true}\rangle$.

—*Step 4.* There are no states with rank ∞.

—*Steps 5-12.* Since Step 4 does not remove any transitions, no deadlock states are created and, hence, the algorithm does not enter the while loop.

—*Step 13.* This step results in the same ranking as in Step 2.

—*Step 14.* Since all reachable states, where $0 \le K$ holds, have a finite rank, the algorithm exits the repeat-until loop.

—*Step 15.* This step removes transitions that start in a low ranking state outside Rank 0 and terminate in a higher rank. For example, the transition $(s_0, s1)$ included in action W_1, where $s_0 = \langle 0, 0, 1, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ and $\langle 0, 1, 0, \mathit{false}, \mathit{false}, \mathit{false}\rangle$, starts in Rank 1 and ends in Rank 2. From s_0, the execution of action QM gets the program to state $\langle 0, 0, 2, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ in Rank 3. Moreover, transitions that form a cycle between the states of the same rank (outside Rank 0) are removed. For instance, the reader R_1 may read the buffer from s_0 and the program reaches the state $s_1 = \langle 1, 0, 1, \mathit{false}, \mathit{true}, \mathit{false}\rangle$. Afterwards, R_1 may take the state of the program back to s_0 by executing the action R_{12}, thereby, creating a cycle between s_0 and s_1 in Rank 1.

The revised program. After applying the Add_UNITY algorithm on the RW program for properties $(0 \leq K) \mapsto (rd_0)$, $(0 \leq K) \mapsto (rd_1)$ and subsequently $(nq > 0) \mapsto (nw = 1)$ the final revised program is as follows:

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

Intuitively, a waiting writer is allowed to write if no other processes have accessed the buffer (i.e., $(wrq) \wedge (K = 3)$). The value of K is zero only if a writer has accessed the buffer, thereby, enabling the writer to release the buffer. The following parameterized action represents the transitions of the reader processes ($j = 0, 1$). A reader process is allowed to read the buffer if no writer is waiting in the internal queue and at most one reader is reading the buffer (i.e., $K > 1$). The guard of the second action has been strengthened in that a reader is allowed to release the buffer if a writer is waiting for access.

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

The behavior of the queue manager process is also modified in that a writer is put in the internal buffer if (1) no writer is currently in the internal buffer, (2) no writer is writing the buffer, and (3) exactly two readers are reading the buffer (i.e., $K = 1$).

$$
QM' : \ (nq = 0)\ \wedge\ (K = 1)\ \wedge\ (nw = 0) \quad \longrightarrow \quad nq := nq + 1;\ wrq := \mathit{true};
$$

We refer the reader to [Ebnenasir et al. 2005] for another example on revising a mutual exclusion algorithm which originally exhibits starvation.

3. REVISING UNTIMED UNITY PROGRAMS: HARDNESS RESULTS

In this section, we present cases where the problem of adding UNITY properties to untimed programs is NP-complete.

3.1 Adding Multiple Progress Properties

In this subsection, we focus on addition of a combination of progress properties (i.e., *leads-to* and/or *ensures*). In this context, we note that the algorithm Add_UNITY can be applied in a stepwise fashion to add multiple progress properties. However, while such stepwise addition is sound, it is not complete. This is due to the fact that during the addition of the first (for instance, *leads-to*) property, the transitions removed in the last step (Line 15 in Algorithm 1) may cause failure in adding subsequent progress properties.

We consider a special case of the problem of adding multiple progress properties where two *eventually* properties are added to a given program. The property 'eventually Q' is logically equivalent with '$true \mapsto Q$' (respectively, *true ensures* Q), i.e., starting from an arbitrary state, the program reaches a state in Q. Thus, for an infinite computation, this implies that Q must be reached infinitely often. Since this special case is NP-complete (see Theorem 3.1 below), the hardness of adding a combination of two *leads-to* and *ensures* properties follows trivially.

Instance. A program $p = \langle S_p, I_p, \delta_p \rangle$ and $SPEC_n \equiv \mathcal{L}_1 \wedge \mathcal{L}_2$, where $\mathcal{L}_1 \equiv (Eventually\ Q)$ and $\mathcal{L}_2 \equiv (Eventually\ T)$, and Q and T are state predicates.

The decision problem (2EV). Given the above instance, does there exist a program $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ such that p' satisfies the constraints of Problem Statement 2.1?

To show the complexity of the above decision problem, we reduce the problem of determining whether or not a directed graph has a simple cycle that includes two specific vertices, described next, to the problem of adding two eventually properties.

Cycle Detection in Directed Graphs (CDDG). Given a directed graph $G = \langle V, A \rangle$, where V is a set of vertices and A is a set of arcs, and two vertices, say u and v in V, does there exist a (simple) cycle in G that includes both u and v? The CDDG problem is known to be NP-complete [Bang-Jensen and Gutin 2002].

Theorem 3.1 *The problem of adding two eventually properties to an untimed program is NP-complete.*

PROOF. Since showing membership to NP is straightforward, we only need to show that the problem is NP-hard. Towards this end, we present a polynomial-time mapping from an instance of CDDG to a corresponding instance of the 2EV problem. Let $G = \langle V, A \rangle$ be a directed graph. We construct $p = \langle S_p, I_p, \delta_p \rangle$ and identify $SPEC_n \equiv \mathcal{L}_1 \wedge \mathcal{L}_2$ as follows:

—$S_p = \{s_x \mid x \in V\}$,
—$\delta_p = \{(s_x, s_y) \mid (x, y) \in A\}$,
—$I_p = \{s_u, s_v\}$,
—$\mathcal{L}_1 \equiv Eventually\{s_u\}$, and $\mathcal{L}_2 \equiv Eventually\{s_v\}$.

Now, we show that the instance of the CDDG problem has a solution if and only if the answer to the corresponding instance of the 2EV problem is affirmative:

—($\Rightarrow$) If the cycle detection problem has a solution then the program obtained by taking only the transitions corresponding to the arcs in that cycle satisfies Problem Statement 2.1.
—($\Leftarrow$) Let $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ be the program obtained after adding two eventually properties. Following the constraints ($C2$) of Problem statement 2.1,

$I_{p'} = \{s_u, s_v\}$. Now, consider a computation of p' that (without loss of generality) starts from s_u. Since p' satisfies *eventually*$\{s_u\}$, state s_u must be revisited in this computation. Consider the smallest prefix where s_u is repeated. For this prefix, we show the following:

(1) State s_v must occur in this prefix. If not, a computation of p' that is obtained by repeating the above computation prefix does not satisfy *eventually*$\{s_v\}$.

(2) No other state can be repeated in this computation prefix. If a state, say s_x, appears twice in the above computation prefix then there would be a cycle between the two occurrences of s_x. This implies that, there is a computation of p' that starts in s_u, reaches s_x and then repeats this cycle. Clearly, this computation does not satisfy *eventually*$\{s_u\}$.

Now, consider the cycle obtained by taking the edges corresponding to the transitions of the above computation prefix. Based on the first point above, this cycle contains both u and v. And, from the second point, it is a simple cycle, i.e., no vertex is repeated in it. $\Box$

Corollary 3.2 *The problem of adding two or more progress properties to untimed UNITY programs is NP-complete.* $\Box$

3.2 Adding a Single *Leads-to* Property with Maximum Non-Determinism

Given a program $p = \langle S_p, I_p, \delta_p \rangle$ and a UNITY specification $SPEC_n$, we say that the revised program p' has *maximum non-determinism* iff $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ meets the constraints of Problem Statement 2.1 and the cardinality of $\delta_{p'}$ is maximum. Maintaining maximum non-determinism is desirable in the sense that it increases the potential for future successful addition of other properties.

Instance. A program $p = \langle S_p, I_p, \delta_p \rangle$, $SPEC_n \equiv (P \mapsto Q)$, and positive integer k, where $k \leq |\delta_p|$.

The decision problem (MND). Given the above instance, does there exist a program $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ such that p' meets the constraints of Problem Statement 2.1 and $|\delta_{p'}| \geq k$?

We now show that the problem of adding a single *leads-to* property while maintaining maximum non-determinism is NP-complete. To this end, we reduce the *feedback arc set problem* in directed graphs to the above decision problem.

Feedback Arc Set Problem (FAS). Let $G = \langle V, A \rangle$ be a digraph and j be a positive integer, where $j \leq |A|$. The feedback arc set problem determines whether there exists a subset $A' \subseteq A$, such that $|A'| \leq j$ and A' contains at least one arc from every directed cycle in G. The FAS problem is known to be NP-complete [Karp 1972].

Theorem 3.3 *The problem of adding a single* leads-to *property while preserving maximum non-determinism is NP-complete.*

PROOF. Since showing membership to NP is straightforward, we only show that the problem is NP-hard. Given an instance of the FAS problem, we present a polynomial-time mapping from FAS instance to a corresponding instance of the MND problem. Let $G = \langle V, A\rangle$ be a directed graph and j be a positive integer. We construct program $p = \langle S_p, I_p, \delta_p\rangle$ and identify integer k and specification $SPEC_n \equiv P \mapsto Q$ as follows:

—$S_p = \{s_v \mid v \in V\} \cup \{p_1, p_2 \cdots p_{|A|+1}\} \cup \{q\}$,
—$I_p = \{p_1, p_2 \cdots p_{|A|+1}\}$,
—$\delta_p = \{(s_u, s_v) \mid (u, v) \in A\} \cup \{(p_i, s_v) \mid (1 \leq i \leq |A| + 1) \wedge (v \in V)\} \cup \{(s_v, q) \mid v \in V\} \cup \{(q, q)\}$, and
—$P = \{p_1, p_2 \cdots p_{|A|+1}\}$, $Q = \{q\}$, and $k = |\delta_p| - j$.

We now show that the instance of FAS has a solution if and only if the answer to the corresponding instance of MND is affirmative:

—($\Rightarrow$) Let the answer to FAS be the set A' of arcs where $|A'| \leq j$. Clearly, given our mapping, constraints ($C1$) and ($C2$) of the Problem Statement 2.1 are met by construction. Now, if we obtain $\delta_{p'}$ by removing the transitions that correspond to A' from δ_p, the resultant program p' will have no cycles in $S_p - (P \cup Q)$. Moreover, since there exists a transition from each state in P to all states in $S_p - (P \cup Q)$ and also there exists a transition from each state in $S_p - P$ to q, any computation that starts from a state in P eventually reaches Q. Observe that the number of transitions removed from δ_p is $|A'|$. Hence, $|\delta_{p'}| = |\delta_p| - |A'| \geq |\delta_p| - j = k$.
—($\Leftarrow$) Let the answer to MND be the program $p' = \langle S_{p'}, I_{p'}, \delta_{p'}\rangle$ where $|\delta_{p'}| \geq k$. We show that the set $A' = \{(x, y) \mid (s_x, s_v) \in \delta_p - \delta_{p'}\}$ is the answer to FAS. Since $(|A| + 1).|V|$ arcs leave states in P, and, the number of transitions that are removed from δ_p (i.e., $|\delta_p - \delta_{p'}|$) is less than $|A|$, any state s_v, where $v \in V$, is reachable from all states in P. Moreover, since $|\delta_{p'}| \geq k = |\delta_p| - j$, it follows that $|A'| = |\delta_p - \delta_{p'}| \leq j$. Now, if there exists a cycle in p, all its transitions must be in the set $\{(s_x, s_y) \mid x, y \in V\}$. Obviously, this cycle is reachable from states in P even though no state in that cycle is in Q. However, this contradicts the assumption that p' satisfies $P \mapsto Q$. Hence, the set of arcs that correspond to transitions in $\delta_p - \delta_{p'}$ (i.e., $|A'|$) contains at least one arc from each cycle in G. $\Box$

4. REVISING REAL-TIME UNITY PROGRAMS: POLYNOMIAL-TIME ALGORITHM

In this section, we focus on automated revision of *real-time* UNITY programs. First, we present the preliminary concepts.

Real-time program. Let $V = \{v_0, v_2 \cdots v_n\}$, $n \geq 0$, be a finite set of *discrete variables* and $X = \{x_0, x_2 \cdots x_m\}$, $m \geq 0$, be a finite set of *clock variables*. Each discrete variable v_i, $0 \leq i \leq n$, is associated with a finite *domain* D_i of values. We assume that the domain of a discrete variable is implicitly encoded with the variable. Thus, if $v_i = v_j$ for some i and j, then $D_i = D_j$ as well. Each clock variable x_j, $0 \leq j \leq m$, ranges over nonnegative real numbers (denoted $\mathbb{R}_{\geq 0}$). A *location* is a function that maps the discrete variables in V to a value from their respective domain. A *clock constraint* over X is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over X by $\Phi(X)$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable x in X. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X.

A *state* s is a pair (l, ν), where l is a location and ν is a clock valuation for X. In this setting, a transition (s_0, s_1) may change either the location or clock valuation of a state and is of the form $(l_0, \nu_0) \rightarrow (l_1, \nu_1)$. Thus, transitions are classified into two types:

—*Immediate transition:* $(l_0, \nu) \rightarrow (l_1, \nu[\lambda := 0])$, where l_0 and l_1 are two locations, ν is a clock valuation, and λ is a set of clock variables, where $\lambda \subseteq X$.

—*Delay transition:* $(l, \nu) \rightarrow (l, \nu + \delta)$, where l is a location, ν is a clock valuation, and $\delta \in \mathbb{R}_{\geq 0}$ is a *time duration*. We denote a delay transition of duration δ at state s by (s, δ).

We define a *real-time program* p in the same fashion that we defined untimed programs in Section 2 (i.e., a tuple $\langle S_p, I_p, \delta_p \rangle$). Note, however, that since clock variables range over real numbers, S_p is an infinite set. Let S be a state predicate of p (i.e., a subset of S_p). We require that if φ is a constraint involving clock variables in X, such that $S \Rightarrow \varphi$, then $\varphi \in \Phi(X)$, i.e., in the corresponding Boolean expression of S, clock variables are only compared to nonnegative integers.

Let $p = \langle S_p, I_p, \delta_p \rangle$ be a real-time program. Also, let δ_p^s and δ_p^d denote the set of immediate and delay transitions of δ_p, respectively, where $\delta_p = \delta_p^s \cup \delta_p^d$. A *computation* of p is an infinite sequence $\sigma = \langle (s_0, t_0), (s_1, t_1) \cdots \rangle$, where states $s_i \in S_p$ and *global time* $t_i \in \mathbb{R}_{\geq 0}$ for all $i \geq 0$, iff (1) $s_0 \in I_p$, (2) $\forall j > 0 : (s_{j-1}, s_j) \in \delta_p$ holds, (3) if σ reaches a terminating state s_f where there does not exist s such that $s \neq s_f$ and $(s_f, s) \in \delta_p^s$ then we extend σ to an infinite computation by stuttering at s_f and letting global time advance, and (4) the sequence $t_0 t_1 \cdots$ satisfies the following constraints:

—*Monotonicity*: For all $i \in \mathbb{Z}_{\geq 0}$, $t_i \leq t_{i+1}$,

—*Divergence*: For all $\tau \in \mathbb{R}_{\geq 0}$, there exists $j \in \mathbb{Z}_{\geq 0}$ such that $t_j \geq \tau$, and

—*Consistency*: For all $i \in \mathbb{Z}_{\geq 0}$, (1) if (s_i, s_{i+1}) is a delay transition (s_i, δ) then $t_{i+1} - t_i = \delta$, and (2) if (s_i, s_{i+1}) is an immediate transition then $t_i = t_{i+1}$.

Real-time UNITY properties [Carruth 1994]. Let P and Q be two arbitrary state predicates.

—(*Bounded-Time Unless*) An infinite timed state sequence $\langle (s_0, t_0), (s_1, t_1) \cdots \rangle$ satisfies P *unless*$_\tau$ Q iff $\forall i \geq 0 : ((s_i \in (P \cap \neg Q)) \Rightarrow \forall j > i \mid (t_j - t_i \leq \tau) : (s_j \in (P \cup Q)))$. Intuitively, if P holds at any state s_i, then for all $j > i$ such that $t_j - t_i \leq \tau$ either (1) Q does not hold in s_j and P is true, or (2) Q becomes true at s_j and P holds at least until Q becomes true. After τ time units there is no requirement on P and Q.

—(*Bounded-Time Leads-to*) An infinite timed state sequence $\langle (s_0, t_0), (s_1, t_1) \cdots \rangle$ satisfies $P \mapsto_\tau Q$ iff if $s_i \in P$, for all $i \geq 0$, then there exists j, $j \geq i$, such that (1) $s_j \in Q$, and (2) $t_j - t_i \leq \tau$. Intuitively, it is always the case that a state in P is followed by a state in Q within τ time units.

The definition of *bounded-time stable* and *bounded-time ensures* are instantiated in the obvious way. The definition of *invariant* is not time-related and remains the same. A *real-time* UNITY *specification* is a conjunction of a set of real-time UNITY properties.

Region graph. Real-time programs can be analyzed with the help of an equivalence relation of finite index on the set of states [Alur and Dill 1994]. Given a real-time program p, for each clock variable $x \in X$, let c_x be the largest constant in the clock constraints of δ_p that involve x, where $c_x = 0$ if x does not occur in any clock constraint of p. We say that two clock valuations ν, μ are *clock equivalent* if:

(1) for all $x \in X$, either $\lfloor \nu(x) \rfloor = \lfloor \mu(x) \rfloor$ or both $\nu(x), \mu(x) > c_x$,

(2) the ordering of the fractional parts of the clock variables in the set $\{x \in X \mid \nu(x) < c_x\}$ is the same in μ and ν, and

(3) for all $x \in X$, where $\nu(x) < c_x$, the clock value $\nu(x)$ is an integer if and only if $\mu(x)$ is an integer.

A *clock region* ρ is a clock equivalence class. Two states (l_0, ν_0) and (l_1, ν_1) are region equivalent, written $(l_0, \nu_0) \equiv (l_1, \nu_1)$, if (1) $l_0 = l_1$, and (2) ν_0 and ν_1 are clock equivalent. A *region* $r = (l, \rho)$ is an equivalence class with respect to $\equiv$, where l is a location and ρ is a clock region. We say that a clock region β is a *time-successor* of a clock region α iff for each $\nu \in \alpha$, there exists $\tau \in \mathbb{R}_{\geq 0}$, such that $\nu + \tau \in \beta$, and $\nu + \tau' \in \alpha \cup \beta$ for all $\tau' < \tau$. We call a region (s, ρ) a *boundary region*, if for each $\nu \in \rho$ and for any $\tau \in \mathbb{R}_{\geq 0}$, ν and $\nu + \tau$ are not equivalent. A region is *open*, if it is not a boundary region. A region (s, ρ) is called an *end region*, if $\nu(x) > c_x$ for all $\nu \in \rho$ and for all clock variables $x \in X$.

Using the region equivalence relation, we construct the *region graph* of a program $p = \langle S_p, I_p, \delta_p \rangle$ (denoted $R(p) = \langle S_p^r, I_p^r, \delta_p^r \rangle$) as follows. Vertices of $R(p)$ (denoted S_p^r) are regions obtained from state space of p. Edges of $R(p)$ (denoted δ_p^r) are of the form $(l_0, \rho_0) \rightarrow (l_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(l_0, \nu_0) \rightarrow (l_1, \nu_1)$ is a transitions in δ_p. Obviously, any set of transition of p transition has a respective set of edges in $R(p)$. A *region predicate* U^r with respect to a state predicate U is defined by $U^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in U \wedge \nu \in \rho)\}$. We say that a region (l_0, ρ_0) of region graph $R(p)$ is a *deadlock region* iff for all regions (l_1, ρ_1) in S_p^r, there does not exist an edge of the form $(l_0, \rho_0) \rightarrow (l_1, \rho_1)$.

We note that region graphs are time-abstract bisimulation of real-time programs [Alur and Dill 1994] and their construction involves an exponential blow-up in the number of clocks and also in the magnitude of clock variables. In our addition algorithm in Subsection 4.1, we transform a real-time program $p = \langle S_p, I_p, \delta_p \rangle$ into its corresponding region graph $R(p) = \langle S_p^r, I_p^r, \delta_p^r \rangle$ by invoking the procedure ConstructRegionGraph as a black box (using the algorithm proposed in [Alur and Dill 1994]). We also let this procedure take state predicates in p (e.g., P and Q) and return the corresponding region predicates in $R(p)$ (e.g., P^r and Q^r). Likewise, we transform a region graph $R(p)$ back into a real-time program by invoking the procedure ConstructRealTimeProgram.

Revised problem statement. The essence of the problem of adding real-time UNITY properties to real-time programs is the same as Problem Statement 2.1. However, in adding real-time UNITY properties, we allow incorporating a finite number of new clock variables. Let $p \backslash T$ denote the program obtained by removing the clock variables in T from the set of clock variables of p. Obviously, no state predicate or set of transitions of $p \backslash T$ depends on the value of variables in T.

Problem Statement 4.1 Given a program $p = \langle S_p, I_p, \delta_p \rangle$, a finite set T of new clock variables, and a real-time UNITY specification $SPEC_n$, identify $p' = \langle S_{p'}, I_{p'}, \delta_{p'} \rangle$ such that:

$(C1)$ $S_{p' \backslash T} = S_p$
$(C2)$ $I_{p' \backslash T} = I_p$
$(C3)$ $\delta_{p' \backslash T} \subseteq \delta_p$
$(C4)$ p' satisfies $SPEC_n$. $\square$

4.1 Adding a Single Bounded-Time *Leads-to* Property

In this subsection, we present a sound and complete algorithm that automatically adds a single bounded-time *leads-to* property to a real-time program. Observe that other real-time UNITY properties can be modeled as a set of transitions. Hence, their addition can be achieved in the same fashion that we did in Subsection 2.2 for untimed programs. Thus, we only focus on addition of bounded-time *leads-to* properties.

Algorithm sketch. Intuitively, the algorithm works in four phases. In Phase 1, we transform the input real-time program into a region graph and subsequently a weighted directed graph (called MaxDelay digraph [Courcoubetis and Yannakakis 1991]). The property of this digraph is such that the longest distance between any two vertices is equal to the maximum time delay between the corresponding regions in the region graph. Then, in Phase 2, we identify a subgraph of the MaxDelay digraph in which the desired bounded-time *leads-to* property is never violated. In Phase 3, we remove deadlock regions. Finally, in Phase 4, we transform the resultant region graph back into a real-time program.

Construction of MaxDelay digraph. We now describe how we transform a region graph $R(p) = \langle S^r_p, I^r_p, \delta^r_p \rangle$ into a MaxDelay digraph $G = \langle V, A \rangle$. Vertices of G consist of the regions in $R(p)$.
Notation: We denote the weight of an arc $(v_0, v_1) \in A$ by $Weight(v_0, v_1)$. Let $f : S^r_p \leftrightarrow V$ denote a bijection that maps each region $r \in S^r_p$ to its corresponding vertex in G and vice versa, i.e., $f(r)$ is a vertex of G that represents region r of $R(p)$ and $f^{-1}(v)$ is the region of $R(p)$ that corresponds to vertex v in V. Let $F : 2^{S^r_p} \leftrightarrow 2^V$ be a bijection that maps a region predicate in $R(p)$ to the corresponding set of vertices of G and let F^{-1} be its inverse. Finally, for a boundary region r with respect to clock variable x, we denote the value of x by $r.x$ (equal to some nonnegative integer).
Arcs of G consist of the following:

—Arcs of weight 0 from v_0 to v_1, if $f^{-1}(v_0) \rightarrow f^{-1}(v_1)$ represents an immediate

Algorithm 2 Add_rtUNITY

Input: real-time program $\langle S_p, I_p, \delta_p \rangle$ and bounded *leads-to* property $P \mapsto_\tau Q$.
Output: revised program $\langle S_{p'}, I_{p'}, \delta_{p'} \rangle$.

Let $c_t := \tau$ where t is a new clock variable; ▷ Phase 1
$\forall((l_0, \nu) \to (l_1, \nu[\lambda := 0])) \in \delta_p \mid (l_0 \notin P \wedge l_1 \in P) : \lambda := \lambda \cup \{t\}$;
$\langle S_p^r, I_p^r, \delta_p^r \rangle, P^r, Q^r :=$ ConstructRegionGraph($\langle S_p, I_p, \delta_p \rangle, P, Q$);
repeat
  IsQRemoved := *false*;
  $\langle V, A \rangle :=$ ConstructMaxDelayGraph($R(p)$);
  $\langle V', A' \rangle :=$ ConstructSubgraph($\langle V, A \rangle, P^r, Q^r, \tau$); ▷ Phase 2
  $\delta_{p'}^r := \{(r_1, r_2) \in \delta_p^r \mid \ \wedge \ (f(r_1), f(r_2)) \in A' \ \vee$
    $\exists r_0 : Weight(f(r_0), f(r_1)) = 1 - \epsilon\}$;
  while $(\exists r_0 \in S_p^r \ : \ (\forall r_1 \in S_p^r \ : \ (r_0, r_1) \notin \delta_{p'}^r))$ **do** ▷ Phase 3
    if $(r_0 \in Q^r)$ **then**
      IsQRemoved := *true*;
      $Q^r := Q^r - \{r_0\}$;
      $\delta_p^r := \delta_p^r - \{(r, r_0) \mid (r, r_0) \in \delta_p^r\}$;
      break;
    end if
    if $(r_0 \notin I_p^r)$ **then**
      $\delta_{p'}^r := \delta_{p'}^r - \{(r, r_0) \mid (r, r_0) \in \delta_{p'}^r\}$;
    else
      `declare that addition is not possible`;
      exit();
    end if
  end while
until (*IsQRemoved* = *false*);
return ConstructRealTimeProgram($\langle S_p, I_p, \delta_{p'}^r \rangle$); ▷ Phase 4

transition in $R(p)$.

—Arcs of weight $c' - c$ from v_0 to v_1, where $c, c' \in \mathbb{Z}_{\geq 0}$ and $c' > c$, if $f^{-1}(v_0)$ and $f^{-1}(v_1)$ are both boundary regions with respect to clock variable x, such that $f^{-1}(v_0).x = c$, $f^{-1}(v_1).x = c'$, and there is a path in $R(p)$ from $f^{-1}(v_0)$ to $f^{-1}(v_1)$ which does not reset x.

—Arcs of weight $c' - c - \epsilon$ from v_0 to v_1, where $c, c' \in \mathbb{Z}_{\geq 0}$, $c' > c$, and $0 < \epsilon \ll 1$, if (1) $f^{-1}(v_0)$ is a boundary region with respect to clock variable x, (2) $f^{-1}(v_1)$ is an open region whose time-successor $f^{-1}(v_2)$ is a boundary region with respect to clock variable x, (3) $f^{-1}(v_0) \to f^{-1}(v_1)$ represents a delay transition in $R(p)$, and (4) $f^{-1}(v_0).x = c$ and $f^{-1}(v_2).x = c'$.

—Self-loop arcs of weight ∞ at vertex v, if $f^{-1}(v)$ is an end region.

In order to compute the maximum time delay between region predicates P^r and Q^r, it suffices to find the longest distance between $F(P^r)$ and $F(Q^r)$ in G. In our addition algorithm, the procedure ConstructMaxDelayGraph transform a region graph $R(p) = \langle S_p^r, I_p^r, \delta_p^r \rangle$ into a MaxDelay digraph $G = \langle V, A \rangle$ as a black box.

The addition algorithm. We now describe the algorithm Add_rtUNITY in details (see Algorithm 2):

—(*Phase 1*) First, in order to keep track of time elapsed since P has become true in a computation, we add an extra clock variable t to p and reset it on immediate

Procedure 3 ConstructSubgraph

Input: MaxDelay digraph $\langle V, A \rangle$, set of vertices V_p and V_q , and an **integer** τ.
Output: a subgraph $\langle V', A' \rangle$.

1: $V' := V$;
2: $A' := \{\}$;
3: **for each** vertex v in V_p **do**
4:     **if** the length of the shortest path Π from v to V_q is at most τ **then**
5:         $A' := A' \cup \{a \mid a \text{ is on } \Pi\}$;
6:     **end if**
7: **end for**
8: $A' := A' \cup \{(u, v) \in A \mid (\forall w \in V' : (w, u) \notin A') \;\vee\; (u \in V_q)\}$;
9: **return** $\langle V', A' \rangle$;

transitions whose source state is not in P and target state is in P (Lines 1-2). Next, we construct the region graph $R(p) = \langle S^r_p, I^r_p, \delta^r_p \rangle$ (Line 3). We now reduce our problem to the problem of bounding the length of the longest path in ordinary weighted digraphs. Towards this end, we first generate the MaxDelay digraph $\langle V, A \rangle$ (Line 6).

—(*Phase 2*) Next, we invoke the procedure ConstructSubgraph (Line 7) which takes a MaxDelay digraph $\langle V, A \rangle$, an integer τ, and two sets of vertices V_p and V_q as input and generates a subgraph of $\langle V, A \rangle$, namely $\langle V', A' \rangle$, whose length of longest path from every vertex in V_p to V_q is bounded by τ (see Procedure 3). We begin with an empty set of arcs (Line 2). Next, we include arcs that participate in the shortest path from each vertex in V_p to a vertex in V_q provided the length of the path is at most τ (Lines 3-7). Then, we add the rest of the arcs to $\langle V', A' \rangle$ (Line 8) except the ones that originate from a shortest path from V_p to V_q identified in Lines 3-7. After invoking ConstructSubgraph, we transform $\langle V', A' \rangle$ back into a region graph $R(p') = \langle S^r_{p'}, I^r_{p'}, \delta^r_{p'} \rangle$ (Line 8 in Algorithm 2).

—(*Phase 3*) We now remove deadlock regions (created due to pruning of arcs in Phase 2) from $R(p')$ (Lines 9-22). However, we need to consider a special case where a region r_0 in Q^r becomes a deadlock region (Lines 10-15). In this case, it is possible that all the regions along a path that starts from some region, say r, in P^r and end in r_0 become deadlock regions. Hence, our algorithm needs to identify a new path from r to a region in Q^r other than r_0. Thus, in such a case, we remove r_0 from Q^r (Lines 12-13) and rerun the algorithm from scratch. If an initial region becomes a deadlock region, we declare failure (Lines 18-20).

—(*Phase 4*) Finally, we construct and return a real-time program out of the region graph $R(p')$ with revised set of edges $\delta^r_{p'}$ (Lines 24).

Level of non-determinism. In order to increase the level of non-determinism, we may include additional paths whose length is at most τ. However, every time we add a path, we need to test whether or not this path creates new paths of length greater than τ. To this end, we can use one of the algorithms in the literature of graph theory (e.g., [Eppstein 1999]) to find and add the k shortest paths in an ordinary weighted digraph.

Theorem 4.2 *The algorithm Add_rtUNITY is sound and complete.*

PROOF. In order to prove soundness, we show that the outcome of the algorithm meets the constraints of Problem Statement 4.1. Since we do not add new states or transitions, constraints $C1$-$C3$ are trivially satisfied. Moreover, by construction, the algorithm only includes states in P from where all computations can reach a state in Q within τ time units. Finally, since the algorithm removes deadlock regions, it does not introduce new time-convergent behaviors to the input program. Therefore, the output of the algorithm satisfies constraint $C4$ as well.

In order to prove the completeness, we show that if an initial state becomes a deadlock state, this state is deadlocked in all real-time programs that satisfy the constraints of Problem Statement 4.1. Observe that the only states that our algorithm may make unreachable are states in P from where there does not exist a computation that reaches a state in Q within τ. Clearly, such states cannot be present in any program that satisfies the constraints of Problem Statement 4.1. Moreover, if a state, say s_1, in P becomes unreachable by removing all its incoming transitions, it is possible that some other state, say s_0, becomes a deadlock state. Likewise, such a state cannot be present in any program that satisfies the constraints of Problem Statement 4.1. If s_0 is an initial state then our algorithm declares failure. Notice that in this case, there exists no solution to the Problem Statement 4.1. □

Theorem 4.3 *The complexity of* Add_rtUNITY *algorithm is polynomial-time in the size of the input program's region graph.* □

4.2 Example: Real-Time Resource Allocation

We now demonstrate how the algorithm Add_rtUNITY works using an example on a real-time resource allocation program. The program RA consists of two processes, RA_1 and RA_2. Each process needs two steps to complete and each step needs 1 time unit to complete. In the first step, the process submits a request for a resource. In the second step, the process performs an I/O operation using the acquired resource. Also, only one step is being executed at a time. The *timed guarded commands* (also called *timed actions*) of RA are as follows:

$$
\begin{array}{lllll}
RQ_j: & req.j \ \wedge \ (x = 1) & \xrightarrow{\{x\}} & io.j, req.j & := \ true, false; \\
IO_j: & io.j \ \wedge \ (x = 1) & \xrightarrow{\{x\}} & req.j, io.j & := \ true, false; \\
WT: & 0 \leq x \leq 1 & \longrightarrow & \textbf{wait}; &
\end{array}
$$

where $j \in \{1, 2\}$. As can be seen, timed guarded command are associated with a (possibly empty) set of clock variables that get reset by executing the timed guarded command (e.g., the clock variable x). The last action (WT) is a delay action where the program is allowed to advance time by an arbitrary time duration as long as the guard continuously remains true.

Clearly, in RA, each process may keep acquiring a resource and performing I/O operation by an unbounded time duration. However, we would like to ensure that RA_1 performs its I/O operation within 2 time units after acquiring the resource. To this end, we add the bounded-time *leads-to* property $\mathcal{L} \equiv (io.1 \mapsto_2 req.1)$. Based on what the algorithm Add_rtUNITY prescribes, we first need to add a new clock variable t and reset it whenever $io.1$ becomes true. Moreover, we let $c_t = 2$ when generating the region graph (see Figure 1). Next, we add the shortest path (the

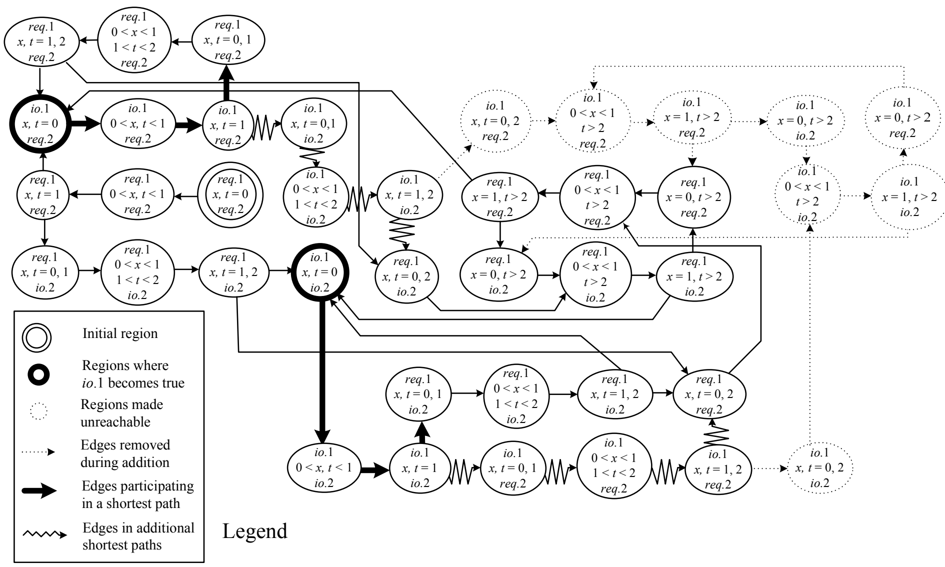

Fig. 1. Region graph of the real-time resource allocation program.

bold edges in Figure 1) from each region where $io.1$ becomes true to a region where $req.1$ holds. Subsequently, we can add additional k shortest paths (the zigzag edges in Figure 1) that preserve $\mathcal{L}$. It is easy to see that the algorithm Add_rtUNITY prunes dotted edges in Figure 1. Also, the regions shown in dotted circles are made unreachable by Add_rtUNITY due to removal of the dotted edges. Thus, the timed guarded commands of the revised program are as follows:

$$
\begin{array}{lll}
RQ_1: & req.1 \ \wedge \ (x = 1) \quad \xrightarrow{\{x,t\}} & io.1, req.1 \ := \ true, false; \\
IO_1: & io.1 \ \wedge \ (x = 1) \quad \xrightarrow{\{x\}} & req.1, io.1 \ := \ true, false; \\
RQ_2: & req.2 \ \wedge \ (x = 1) \ \wedge \ (io.1 \Rightarrow t \leq 1) \quad \xrightarrow{\{x\}} & io.2, req.2 \ := \ true, false; \\
IO_2: & io.2 \ \wedge \ (x = 1) \ \wedge \ (io.1 \Rightarrow t \leq 1) \quad \xrightarrow{\{x\}} & req.2, io.2 \ := \ true, false; \\
WT: & 0 \leq x \leq 1 \quad \longrightarrow & \textbf{wait};
\end{array}
$$

Notice that if we only add the shortest paths from regions where $io.1$ becomes true, i.e., we do not add additional shortest paths, then in the resulting program, the second conjunct in timed actions RQ_2 and IO_2 would be replaced with $io.1 = false$. In this case, we would force the program to always execute the second step of RA_1 (i.e., timed action IO_1) immediately after the first step (i.e., timed action RQ_1).

We refer the reader to [Bonakdarpour and Kulkarni 2006a] for another example on revising a controller for a railroad crossing gate which originally exhibits unbounded wait.

5. REVISING REAL-TIME UNITY PROGRAMS: HARDNESS RESULT

In this section, we show that the problem of adding a *bounded-time leads-to* property to a real-time program while maintaining maximum non-determinism is NP-complete in the size of the program's region graph even if the given program satisfies the corresponding *unbounded leads-to* property.

Instance. Region graph $R(p) = \langle S_p^r, I_p^r, \delta_p^r \rangle$ of a real-time program p, a *bounded-time leads-to* property $\mathcal{L} \equiv (P \mapsto_\tau Q)$, and a positive integer k, where p satisfies $P \mapsto Q$ and $|\delta_p^r| \geq k$.

The decision problem (MNBL). Given the above instance, does there exist a region graph $R(p') = \langle S_{p'}^r, I_{p'}^r, \delta_{p'}^r \rangle$, such that $|\delta_{p'}^r| \geq k$ and $R(p')$ meets the constraints of Problem Statement 4.1?

We prove that MNBL is NP-complete by a reduction from the *vertex splitting problem* [Paik et al. 1994; 1998] in weighted directed acyclic graphs (DAG) described next.

The DAG Vertex Splitting Problem (DVSP). Let $G = \langle V, A \rangle$ be a weighted DAG and v_{sc}, v_{tg} be unique source and target vertices in V where the indegree of v_{sc} and the outdegree of v_{tg} are zero. Let G/Y denote the DAG when each vertex $v \in Y$ is split into vertices v^{in} and v^{out} such that all arcs $(v, u) \in A$, where $u \in V$, are replaced by arcs of the form (v^{out}, u) and all arcs $(w, v) \in A$, where $w \in V$, are replaced by arcs of the form (w, v^{in}). In other words, the outgoing arcs of v now leave vertex v^{out} while the incoming arcs of v now enter v^{in}, and, there is no arc between v^{in} and v^{out}. The DAG vertex splitting problem is to find a vertex set Y, where $Y \subseteq V$ and a positive integer i, where $|Y| \leq i$, such that the length of the longest path of G/Y from v_{sc} to v_{tg} is bounded by a pre-specified value d. DVSP is known to be NP-complete [Paik et al. 1994; 1998], for the case where $d \geq 2$ and the weight of all arcs is 1.

Theorem 5.1 *The problem of adding a* bounded-time leads-to *property to a real-time program is NP-complete in the size of the program's region graph even if the program satisfies the corresponding* unbounded leads-to *property.*

Proof. Since membership to NP is trivial, we show that the problem is NP-hard.
Mapping. Let $G = \langle V, A \rangle$ be any instance of DVSP whose longest path is to be bounded by d. We construct now a real-time program MP (and as a result the region graph $R(MP) = \langle S_{MP}^r, I_{MP}^r, \delta_{MP}^r \rangle$) in the form of timed guarded commands as follows. Let the set of discrete and clock variables of MP be the singletons $\{l\}$ and $\{x\}$, respectively. For each vertex $v \in V - \{v_{tg}\}$ and for the target vertex v_{tg}, we introduce the following timed guarded commands:

$$
\begin{array}{lllllll}
MP_{v.1}: & (l = v^{in}) & \wedge & (x = 0) & \wedge\ (l \neq v_{tg}^{in}) & \longrightarrow & l := v^{out}; \\
MP_{v.2}: & (l = v^{in}) & \wedge & (x = 0) & \wedge\ (l \neq v_{tg}^{in}) & \longrightarrow & l := v_{tg}^{out}; \\
MP_{v_{tg.n.1}}: & (l = v_{tg}^{in}) & \wedge & (x = 0) & & \longrightarrow & l := tmp_{tg.n}; \\
MP_{v_{tg.n.2}}: & (l = tmp_{tg.n}) & \wedge & (x = |A| + 1 - d) & & \xrightarrow{\{x\}} & l := v_{tg}^{out}; \\
MP_{v_{tg}}: & (l = v_{tg}^{out}) \vee & & & & & \\
 & ((l = tmp_{tg.n}) & \wedge & (0 \leq x \leq |A| + 1 - d)) & & \longrightarrow & \mathbf{wait};
\end{array}
$$

for all $1 \leq n \leq 2|V|$. For the source vertex v_{sc}, we let $v_{sc}^{in} = v_{tg}^{out}$. Also, for each arc (u, v) in A, we introduce the following timed guarded commands to MP:

$$
\begin{array}{llclcl}
MP_{(u,v).j.1}: & (l = u^{out}) & \wedge & (x = 0) & \longrightarrow & l := tmp_{(u,v).j}; \\
MP_{(u,v).j.2}: & (l = tmp_{(u,v).j}) & \wedge & (x = 1) & \xrightarrow{\{x\}} & l := v^{in}; \\
MP_{(u,v)}: & (l := tmp_{(u,v).j}) & \wedge & (0 \leq x \leq 1) & \longrightarrow & \textbf{wait};
\end{array}
$$

for all j, where $1 \leq j \leq 2|V|$. Intuitively, for each arc $(u, v) \in A$, the discrete variable l in program MP is assigned one of the following values: v^{in}, v^{out}, u^{in}, u^{out}, or $tmp_{(u,v).1} \cdots tmp_{(u,v).2|V|}$. Clearly, the value of l along with the clock regions identify S^r_{MP}. The set of initial regions of $R(MP)$ is the singleton $I^r_{MP} = \{(l = v^{in}_{sc}, x = 0)\}$. The set δ^r_{MP} of edges is identified by the above set of timed guarded commands. Finally, we let $P = \{s \mid l(s) = v^{in}_{sc}\}, Q = \{s \mid l(s) = v^{out}_{tg}\}$, $k = |\delta^r_{MP}| - i$, and $\tau = |A| + 1$. Observe that all computations of MP start from where P holds and eventually reach Q, as G is acyclic. Hence, MP satisfies $P \mapsto Q$. We note that since v_{sc} and v_{tg} are unique vertices in G, Q is reachable from all states in MP and, hence, MP satisfies $true \mapsto Q$ as well.

Reduction. We need to show that a vertex $v \in Y$ in G must be split iff the corresponding timed guarded command $(l = v^{in}) \wedge (x = 0) \wedge (l \neq v^{in}_{tg}) \rightarrow l := v^{out}$ must be removed from MP:

—($\Rightarrow$) Let the answer to DVSP be the set Y, where $|Y| \leq i$, i.e., after splitting all vertices $v \in Y$, the length of the longest path in G is at most d. We obtain the region graph $R(MP') = \langle S^r_{MP'}, I^r_{MP'}, \delta^r_{MP'}\rangle$ as follows. First, we let $S^r_{MP'} = S^r_{MP}$ and $I^r_{MP'} = I^r_{MP}$. In order to obtain $\delta^r_{MP'}$, we remove the edges that correspond to timed action $MP_{v.1}$ from δ^r_{MP}, for all $v \in Y$. Since v_{tg} is the unique target vertex in G, Q remains to be the set $\{s \mid l(s) = v^{out}_{tg}\}$ in MP'. Thus, any computation of MP' that begins from a state in P will reach Q. Now, we show that the maximum time delay to reach Q is τ. Observe that there are two ways to reach Q: (1) from the state where $l = v^{in}_{tg}$ (using timed actions $MP_{v_{tg.n.1}}$ and $MP_{v_{tg.n.2}}$ for some n, $1 \leq n \leq 2|V|$), and (2) from a state where $l \neq v^{in}_{tg}$ (using the immediate transition in timed action $MP_{v.2}$ for some $v \in V$). In the former case, the delay in reaching the state where $l = v^{in}_{tg}$ is less than d and since the time delay of timed actions $MP_{v_{tg.n.1}}$ and $MP_{v_{tg.n.2}}$ is $|A| + 1 - d$, the total time delay to reach Q is at most $|A| + 1 = \tau$. In the latter case, by construction, the delay to reach Q is at most τ. Moreover, recall that $k = |\delta^r_{MP}| - i$. Therefore, MP' meets the constraints of Problem Statement 4.1 with respect to $\mathcal{L}$ and $|\delta^r_{MP'}| \geq |\delta^r_{MP}| - i = k$.

—($\Leftarrow$) Let the answer to MNBL be $R(MP') = \langle S^r_{MP'}, I^r_{MP'}, \delta^r_{MP'}\rangle$, where $|\delta^r_{MP'}| \geq k$ and the maximum delay to reach Q from P is at most τ. Note that $I^r_{MP'}$ must be $\{(l = v^{in}_s, x = 0)\}$. Observe that in order to obtain $R(MP')$, removing one or more timed guarded commands $MP_{(u,v).j.1}$, $MP_{(u,v).j.2}$, $MP_{v_{tg.n.1}}$, or $MP_{v_{tg.n.2}}$ does not contribute in bounding the maximum delay. This is due to the fact that the number of edges removed from δ^r_{MP} is at most $|\delta^r_{MP}| - k$, and $k = |\delta^r_{MP}| - i$, where $i \leq |V|$, and there are $2|V|$ of such guarded commands and, hence, their removal would not change the maximum delay. Thus, we can assume that the edges removed are of the form $(l = v^{in}) \wedge (x = 0) \wedge (l \neq v^{in}_{tg}) \rightarrow l := v^{out}$. Observe that in order to reach Q from P, a computation either takes a timed guarded command $MP_{v.2}$ for some $v \in V$, or it reaches Q via the state where $l = v^{in}_{tg}$. Clearly, in the later case, the delay to reach the state where $l = v^{in}_{tg}$ is at most $\tau - (|A| + 1 - d) = d$. In the former case, the corresponding path in G does not exist and, hence, its length does not matter. Thus, the timed actions removed to obtain $\delta^r_{MP'}$ identify the set Y of vertices that should be split in G, i.e, $Y = \{v \in V - \{v_{tg}\} \mid ((l = v^{in}, x = 0) \rightarrow (l = v^{out}, x = 0)) \in (\delta^r_{MP} - \delta^r_{MP'})\}$. $\square$

6. RELATED WORK

In this section, we illustrate how the contributions of this paper differ from existing approaches for program synthesis and transformation. 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) [Emerson and Clarke 1982; Manna and Wolper 1984; Pnueli and Rosner 1989a; 1989b; Attie et al. 2004; Attie 1999; Attie and Emerson 2001], 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 algorithms 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. In this context but from a game theoretical perspective, Jobstmann, Griesmayer, and Bloem [Jobstmann et al. 2005] independently show that the problem of repairing a program with respect to two Büchi conditions in NP-complete.

The algorithms for automatic addition of fault-tolerance [Kulkarni et al. 2007; Kulkarni et al. 2001; Kulkarni and Ebnenasir 2002; 2003; 2004; Bonakdarpour and Kulkarni 2006b] 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 [Kulkarni et al. 2007; Kulkarni et al. 2001; Kulkarni and Ebnenasir 2003; 2004; Bonakdarpour and Kulkarni 2006b] to add fault-tolerance concerns to a UNITY program synthesized by the algorithm presented in this paper.

Algorithms for comprehensive redesign of timed automata [Alur and Dill 1994] from real-time temporal logic MITL formulae was first introduced in [Alur et al. 1996]. More recently, in [Maler et al. 2006], the authors present much simpler algorithms for constructing timed automata from MITL formulae than the ones in [Alur et al. 1996].

Synthesis of real-time systems has mostly been studied in the context of controller synthesis and game theory [Asarin et al. 1998; Asarin and Maler 1999; D'Souza and Madhusudan 2002; Bouyer et al. 2003; de Alfaro et al. 2003; Faella et al. 2002]. In these papers, the common assumption is that the existing program (called a *plant* in controller synthesis and a *game* in game theory) and/or the given specification are *deterministic*. In these papers, since the authors consider highly expressive specifications, the complexity of the proposed methods is very high. For example, synthesis problems presented in [Faella et al. 2002; Asarin et al. 1998; Asarin and

Maler 1999; de Alfaro et al. 2003] are EXPTIME-complete. Moreover, deciding the existence of a solution (called a controller) in [D'Souza and Madhusudan 2002; Bouyer et al. 2003] is 2EXPTIME-complete. Both complexity classes are in the size of the given plant or game. By contrast, the complexity of our algorithms is significantly less.

7. CONCLUSION AND FUTURE WORK

We focused on the problem of revising UNITY [Chandy and Misra 1988] programs where one adds a conjunction of UNITY properties (*unless, stable, invariant, ensures* and *leads-to*) to an existing program while preserving the existing UNITY properties. Intuitively, in our approach for such revision, only behaviors that violate a UNITY property are removed. This ensures that even unknown existing UNITY properties continue to be satisfied. In this context, a sound and complete revision method is highly valuable since it allows the designer to determine whether the given program is *fixable* and if it is fixable, it provides a new program that satisfies the desired UNITY property. In particular, we focused on the complexity of revising programs with respect to UNITY properties. We identified cases where polynomial-time solution is possible and cases where the problem is NP-complete.

Regarding polynomial-time addition of UNITY properties, we showed that any conjunction of *unless, stable*, and *invariant* properties along with a single *leads-to* (respectively, *ensures*) property can be added in polynomial-time. We also showed that for real-time UNITY programs, it is possible to add a single *bounded-time leads-to* property along with (a conjunction of) any number of safety properties.

We also found a surprising result that while the problem of adding a single *leads-to* is simple, the problem becomes hard even with small changes. In particular, we showed that the problem of adding two *leads-to* properties (respectively, any combination of *leads-to* and *ensures*) is NP-complete. Moreover, if we desire maximum non-determinism, i.e., we want to maximize the number of program transitions of the synthesized program, then the problem of adding even a single *leads-to* property is NP-complete. These two NP-completeness results hold for both untimed and timed addition of UNITY programs. Therefore, to study the possible differences between revising untimed UNITY programs and timed UNITY programs, we focused on the case where the initial timed program satisfies $true \mapsto Q$. Then, it trivially satisfies the untimed property $P \mapsto Q$. Thus, providing maximum non-determinism for adding untimed property $P \mapsto Q$ is trivial. However, even for this case, the problem of providing maximum non-determinism for adding a bounded-time *leads-to* property is NP-complete.

To extend the results of this paper, we plan to integrate our algorithms with existing model checkers 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. In this regard, we also intend to identify constraints on the initial program and added properties so that addition of multiple *leads-to* properties can be achieved in polynomial-time.

We also plan to incorporate symbolic techniques in our algorithms and integrate them with our tool SYCRAFT [Bonakdarpour and Kulkarni 2008] for adding

UNITY properties. In particular, in [Bonakdarpour and Kulkarni 2007], we have shown that symbolic techniques can be used to effectively synthesize moderate-sized distributed programs (reachable states of size 10^{50}). We, therefore, expect that these approaches would also assist significantly in managing state space during the addition of UNITY properties.

REFERENCES

Alur, R. and Dill, D. 1994. A theory of timed automata. *Theoretical Computer Science 126,* 2, 183–235.

Alur, R., Feder, T., and Henzinger, T. 1996. The benefits of relaxing punctuality. *Journal of the ACM 43,* 1, 116–146.

Asarin, E. and Maler, O. 1999. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*. 19–30.

Asarin, E., Maler, O., Pnueli, A., and Sifakis, J. 1998. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*. 469–474.

Attie, P. and Emerson, E. A. 2001. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23,* 2, 187 – 242.

Attie, P. C. 1999. Synthesis of large concurrent programs via pairwise composition. In *International Conference on Concurrency Theory (CONCUR)*. Springer-Verlag, London, UK, 130–145.

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

Bang-Jensen, J. and Gutin, G. 2002. *Digraphs: Theory, Algorithms and Applications*. Springer.

Bonakdarpour, B. and Kulkarni, S. S. 2006a. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. LNCS 4346. 261–276.

Bonakdarpour, B. and Kulkarni, S. S. 2006b. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. LNCS 4280. 122–136.

Bonakdarpour, B. and Kulkarni, S. S. 2007. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 3–10.

Bonakdarpour, B. and Kulkarni, S. S. 2008. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*. 167–171.

Bouyer, P., D'Souza, D., Madhusudan, P., and Petit, A. 2003. Timed control with partial observability. In *Computer Aided Verification (CAV)*. 180–192.

Carruth, A. 1994. Real-time UNITY. Tech. Rep. CS-TR-94-10, University of Texas at Austin. January.

Chandy, K. M. and Misra, J. 1988. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Courcoubetis, C. and Yannakakis, M. 1991. Minimum and maximum delay problems in real-time systems. In *Computer-Aided Verificaion (CAV)*. 399–409.

de Alfaro, L., Faella, M., Henzinger, T. A., Majumdar, R., and Stoelinga, M. 2003. The element of surprise in timed games. In *International Conference on Concurrency Theory (CONCUR)*.

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

Dijkstra, E. W. 1990. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

D'Souza, D. and Madhusudan, P. 2002. Timed control synthesis for external specifications. In *Symposium on Theoretical Aspects of Computer Science (STACS)*. 571–582.

Ebnenasir, A., Kulkarni, S. S., and Bonakdarpour, B. 2005. Revising UNITY programs: Possibilities and limitations. In *On Principles of Distributed Systems (OPODIS)*. 275–290.

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

Eppstein, D. 1999. Finding the k shortest paths. *SIAM Journal of Computing 28,* 2, 652–673.

Faella, M., LaTorre, S., and Murano, A. 2002. Dense real-time games. In *Logic in Computer Science (LICS)*. 167–176.

Holzmann, G. 1997. The model checker spin. *IEEE Transactions on Software Engineering*.

Jobstmann, B., Griesmayer, A., and Bloem, R. 2005. Program repair as a game. In *Computer Aided Verification (CAV)*. 226–238.

Karp, R. M. 1972. Reducibility among combinatorial problems. In *Symposium on Complexity of Computer Computations*. 85–103.

Kulkarni, S. S., Arora, A., and Chippada, A. 2001. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*. 130–140.

Kulkarni, S. S., Arora, A., and Ebnenasir, A. 2007. *Software Engineering and Fault-Tolerance*. World Scientific Publishing Co. Pte. Ltd, Chapter Adding Fault-Tolerance to State Machine-Based Designs.

Kulkarni, S. S. and Ebnenasir, A. 2002. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems (ICDCS)*, 337–344.

Kulkarni, S. S. and Ebnenasir, A. 2003. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*.

Kulkarni, S. S. and Ebnenasir, A. 2004. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*. 209–219.

Lafortune, S. and Lin, F. 1992. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications 1,* 1, 61–92.

Lin, F. and Wonham, W. M. 1990. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control 35,* 12 (December).

Maler, O., Nickovic, D., and Pnueli, A. 2006. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*. 274–289.

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

Paik, D., Reddy, S., and Sahni, S. 1994. Deleting vertices to bound path length. *IEEE Transations on Computers 43,* 9, 1091–1096.

Paik, D., Reddy, S. M., and Sahni, S. 1998. Vertex splitting in dags and applications to partial scan designs and lossy circuits. *International Journal of Foundations of Computer Science 9,* 4, 377–398.

Pnueli, A. and Rosner, R. 1989a. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*. 179–190.

Pnueli, A. and Rosner, R. 1989b. On the synthesis of an asynchronous reactive module. In *International Colloqium on Automata, Languages, and Programming (ICALP)*. 652–671.

Ramadge, P. and Wonham, W. 1989. The control of discrete event systems. *Proceedings of the IEEE 77,* 1, 81–98.

Rohloff, K. R. 2004. Computations on distributed discrete-event systems. Ph.D. thesis, University of Michigan.

Rudie, K., Lafortune, S., and Lin, F. 2003. Minimal communication in a distributed discrete-event systems. *IEEE Transactions On Automatic Control 48,* 6 (June).

Thomas, W. 2002. Infinite games and verification (extended abstract of a tutorial). In *International Conference on Computer Aided Verification (CAV)*. 58–64.

Wallmeier, N., Hütten, P., and Thomas, W. 2003. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*. 11–22.