Computer Science Technical Report

Identifying Satisfying Subsets: A Method for Algorithmic Correction of Inter-Thread Synchronization Mechanisms

Ali Ebnenasir
Mohammad Amin Alipour

Michigan Technological University
Computer Science Technical Report
CS-TR-10-01
March 2010

This work was partially sponsored by the NSF grant CCF-0950678.

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Identifying Satisfying Subsets: A Method for Algorithmic Correction of Inter-Thread Synchronization Mechanisms

Ali Ebnenasir
Mohammad Amin Alipour

March 2010

This work was partially sponsored by the NSF grant CCF-0950678.

Abstract

Correcting concurrency failures (e.g., data races and deadlocks/livelocks) is a difficult task in part due to the non-deterministic nature of multithreaded programs, and the fact that a correction often involves multiple threads. As such, manually fixing a failure often results in introducing new flaws, which in turn requires a regression verification step after every fix. Most existing methods for detection and correction of data races rely on code annotation (e.g., atomic sections) or linguistic constructs (e.g., coordination models) that enable programmers to introduce non-determinism in a controlled fashion at the expense of degraded performance. In this report, we present an alternative paradigm for the correction of data races and *non-progress* failures, where we systematically search for a *subset* of inter-thread synchronization traces of faulty programs that is data race-free and meets progress requirements, called a *satisfying subset* of traces. To evaluate the effectiveness of satisfying subsets in the correction of concurrency failures, we have developed explicit-state and symbolic implementations of our algorithms. Our software tools have facilitated the correction of data races and non-progress failures in several Unified Parallel C (UPC) and Java programs that programmers have failed to correct in a reasonable amount of time.

Contents

1 Introduction

Designing and debugging inter-thread synchronization mechanisms in multithreaded programs are complex tasks in part due to the inherent non-determinism and the crosscutting nature of data race-freedom and progress (i.e., deadlock/livelock-freedom). As such, manually fixing concurrency failures[1] often results in introducing new design flaws, which in turn requires a regression verification step after every fix. Thus, given a program that has data races or non-progress failures for some threads, it is desirable to automatically correct the failures while preserving the correctness of other threads. The motivation behind such automation is multi-fold. An algorithmic correction method (i) provides a systematic way for pruning (unnecessary) non-determinism; (ii) facilitates designing and debugging of multithreaded programs; (iii) separates synchronization from functional concerns; (iv) reuses the computations of an existing program towards fixing it, thereby potentially preserving its quality of service (e.g., performance), and (v) preserves the liberty of programmers in designing/controlling concurrency using existing synchronization primitives (e.g., locks).

Numerous approaches exist for detection/correction of concurrency failures, most of which have little support for automatic correction of non-progress failures and require new linguistic constructs (e.g., atomic sections). For example, several researchers present techniques for automatic detection of low-level [38] and high-level [8, 15, 40, 28] data races. Many researchers use type systems [27, 25, 26] to enable atomicity and to detect atomicity violations [18]. Automatic locking techniques [36, 17] generate lock-based synchronization functionalities from annotated code to ensure data race-freedom at the expense of limiting programmers' control over thread concurrency and performance. Atomic blocks [5] and software transactional memory [30, 19] present optimistic methods based on the concept of atomic transactions, where a concurrent access to the shared data causes a roll-back for a transaction. *Specification-based* synthesis approaches [24, 37, 20, 32] generate synchronization functionalities from formal specifications with little chance of reuse if new requirements are added to an existing program. Wang *et al.* [43] apply control-theoretic methods for deadlock avoidance, nonetheless, it is unclear how their approach could be used for correcting livelocks/starvations. Program repair methods [32] use game-theoretic techniques to synthesize correct synchronization mechanisms from formal specifications. Vechev *et al.* [41] infer synchronization mechanisms from safety requirements. While the aforementioned approaches inspire our work, they often lack a systematic method for the *correction* of both data races and non-progress failures.

We propose an alternative paradigm, where we start with a program with data races and/or non-progress failures, and we systematically search for a *subset* of its inter-thread synchronization traces that is data race-free and ensures deadlock/livelock-freedom, called a *Satisfying Subset* (SS) of traces. Figure 1 illustrates this paradigm. We initially perform a static analysis on the faulty program to generate a slice of it in Promela [31, 4][2] representing a projection of program behaviors on synchronization variables. We call the sliced program, the *synchronization skeleton*. Then we systematically generate a behavioral model of the synchronization skeleton represented as either a reachability graph or a symbolic model in terms of Binary Decision Diagrams (BDDs) [14]. Subsequently, we use two families of algorithms: one for finding data race-free SSs and the other for identifying SSs that meet progress requirements. If there is a SS, then our algorithms generate a revised version of the synchronization skeleton (in Promela). We then translate the corrections back to the level of source code.

In order to evaluate the effectiveness of satisfying subsets in automatic correction of concurrency failures, we have developed explicit-state and symbolic implementations of our correction algorithms. As we illustrate in Section 8, our experience shows that a hybrid use of different data structures provides better scalability. Using our software tools, we have corrected data races and non-progress failures in several Java and Unified Parallel C (UPC) programs that programmers have failed to correct in a reasonable amount of time. For instance, the symbolic implementation of our approach enabled the correction of starvation in a barrier synchronization protocol with 150 threads (with almost 3^{150} reachable states) on a regular PC in half an hour.

The **contributions** of this report are as follows: We apply our previous work [21, 12] on the theory of program revision for Linear Temporal Logic (LTL) [23] properties to two real-world programming languages,

[1] We follow the dependability community [9] in calling data races and deadlocks/livelocks *failures* since they are manifestations of design flaws/faults in program behaviors.

[2] The choice of Promela as an intermediate language is to benefit from verification and visualization features of SPIN in debugging.

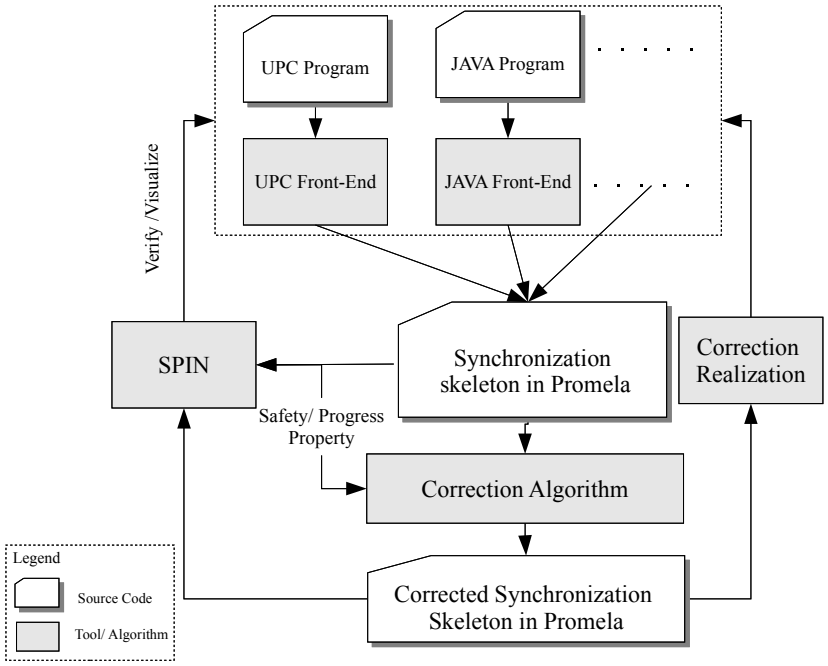

Figure 1: Overall view of Satisfying Subset approach.

namely UPC and Java. Specifically, we enable the algorithmic correction of a classic family of data races in legacy programs, where a *data race* occurs when multiple threads access a shared datum concurrently and at least one of the threads performs a write operation [38]. Moreover, we present an improved version of the algorithm presented in [21] for correction of non-progress failures. Our improved algorithm illustrates an average 27% run-time improvement in the experiments we have conducted thus far. More importantly, our algorithm corrects non-progress failures without any fairness assumptions. This is a significant contribution as experimental evidence [1, 2, 3] points to failures that may manifest themselves only on some schedulers due to different fairness policies implemented by schedulers. Moreover, recent research illustrates that implementing fair schedulers is a hard problem [34] (and sometimes even impossible). As a testimony to the significance of this problem, we quote Joshua Bloch (see Pages 286-287 of [11]) on the Java thread scheduler: "*Do not depend on the thread scheduler for the correctness of your program. The resulting program will be neither robust nor portable.*" As such, we enable systematic correction of non-progress failures in a scheduler-independent manner; i.e., corrected programs are portable.

Organization. The organization of the report is as follows: Section 2 presents preliminary concepts. Section 3 states the problem of algorithmic correction. Then, Section 4 discusses how we extract the synchronization skeleton of programs. Section 5 presents an algorithm for the correction of non-progress failures, and Section 6 focuses on the correction of data races. Section 7 presents thress example for fixing safety violation and non-progress property in programs.Subsequently, Section 8 presents our experimental results. A discussion on the limitations of our approach appears in Section 9. Finally, we make concluding remarks and discuss future work in Section 10.

2 Preliminaries

In this section, we present formal definitions of synchronization skeletons of multi-threaded programs, safety and leads-to properties, specifications (adapted from [7]) and the fairness assumption that we consider in

this report.

Synchronization skeletons. Consider a multithreaded program p with a fixed number of threads, denoted N. The synchronization skeleton of p is a non-deterministic finite state machine denoted by a triple $\langle V_p, \delta_p, I_p \rangle$, where V_p represents a finite set of synchronization variables with finite domains. A *synchronization state* is a unique valuation of synchronization variables. Throughout this report, we use the terms *state* and *synchronization state* interchangeably. An ordered pair of states (s_0, s_1) denotes a transition. A *thread* contains a set of transitions, and δ_p denotes the union of the set of transitions of threads of p. I_p represents a set of initial states. The state space of p, denoted $\mathcal{S}_p$, is equal to the set of all states of p. A *state predicate* is a subset of $\mathcal{S}_p$. A state predicate $\mathcal{X}$ is true (i.e., holds) in a state s iff (if and only if) $s \in \mathcal{X}$. A *computation* (i.e., *synchronization trace*) is a *maximal* sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states s_i, where each transition (s_i, s_{i+1}) is in δ_p, $i \geq 0$. That is, either σ is infinite, or if σ is a finite sequence $\langle s_0, s_1, \cdots, s_f \rangle$, then no transition in δ_p originates at s_f. A *computation prefix* is a *finite* sequence $\sigma = \langle s_0, s_1, \cdots, s_k \rangle$ of states, where each transition (s_i, s_{i+1}) is in δ_p, $0 \leq i < k$.

Example: Swapping program. As an example, consider an excerpt of a Unified Parallel C (UPC) program in Figure 2. The UPC language [22] is an extension of the C programming language that supports the Single Program Multiple Data (SPMD) model in a Partitioned Global Address Space (PGAS) shared memory model. Several groups of researchers (e.g., Berkeley's Lawrence Lab) use UPC on a variety of cluster [10] and multicore [6] platforms to implement and run High Performance Computing (HPC) applications.

Consider a *shared* integer array `A` of `THREADS` elements and `THREADS` UPC threads (Line 1 in Figure 2). In UPC, processes are called *threads* and the system constant `THREADS` is the number of threads. Each thread's own thread number is denoted `MYTHREAD`. Shared data structures are explicitly declared with a `shared` type modifier. Each thread repeatedly initializes `A[MYTHREAD]` and randomly chooses an array element to swap with the contents of `A[MYTHREAD]`. A shared array of `THREADS` locks `lk` is declared in Line 2. Each thread T_i uses `lk`$[i]$ and `lk`$[s]$ to lock the two elements it wants to swap so no data races occur. A block of instructions accessing shared data is called a *critical section* of the code (Lines 12-14 in Figure 2). The section of the code where a thread is trying to acquire the locks for entering its critical section is called a *trying section* (Lines 10-11 in Figure 2). Section 4 explains how we extract the synchronization skeleton of the swapping program (illustrated in Listing 1).

```
1  shared int A[THREADS];
2  upc_lock_t *shared lk[THREADS];
3  int i, s, temp;
4
5  A[MYTHREAD] = MYTHREAD;
6  i = MYTHREAD; // Variable i is a thread index.
7  for (;;) { // For loop
8  // Randomly generate an index to swap with
9  s = (int)lrand48() % (THREADS);
10   upc_lock(lk[i]);  // Acquire the necessary locks
11   upc_lock(lk[s]);
12 temp = A[i];  // Swap
13 A[i] = A[s];
14 A[s] = temp;
15 upc_unlock(lk[s]);  // Unlock
16 upc_unlock(lk[i]);
17 }  // For loop
```

Figure 2: Swapping program in UPC.

Properties and specifications. Intuitively, a *safety* property stipulates that nothing bad ever happens. Formally, we represent a safety property as a state predicate that must always be true in program computations, called *invariance* properties. In the UPC program of Figure 2, a safety property stipulates that *it is always the case* that at most one thread is in its critical section. We formally specify such properties using the *always* operator in LTL [23], denoted $\Box$. The example UPC code ensures that the safety property $\Box\neg(CS_0 \wedge CS_1)$ is met by acquiring locks, where CS_i $(i = 0, 1)$ is a state predicate representing that thread i is in its critical section, where THREADS$= 2$.

A *progress* property states that *it is always the case that if P becomes true in some program computation, then Q will eventually hold in that computation*, where P and Q are state predicates. We denote such progress properties by $\mathcal{P} \rightsquigarrow \mathcal{Q}$ (read it as 'P leads to Q') [23]. For example, in the swapping program of Figure 2, we specify progress for each thread i ($i = 0, 1$) as $TS_i \rightsquigarrow CS_i$; i.e., *it is always the case that if thread i is in its trying section, denoted by a state predicate TS_i, then it will eventually enter its critical section CS_i*. We define a *specification spec* as an intersection of a set of safety and leads-to properties.

A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ *satisfies* (i.e., *meets*) a safety property $\Box S$ from an initial state s_0 iff the state predicate S holds in s_i, for all $i \geq 0$. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ *satisfies* (i.e., *meets*) a leads-to property $P \rightsquigarrow Q$ from an initial state s_0 iff for any state s_j, where $j \geq 0$, if P holds in s_j, then there exists a state s_k, for $k \geq j$, such that Q is true in s_k. A program $p = \langle V_p, \delta_p, I_p \rangle$ meets a property $\mathcal{L}$ from I_p iff all computations of p meet $\mathcal{L}$ from I_p. A program $p = \langle V_p, \delta_p, I_p \rangle$ meets its specification *spec* from I_p iff p meets all properties of *spec* from I_p. Whenever it is clear from the context, we abbreviate 'p meets *spec* from I_p' by 'p meets *spec*'.

Non-progress and fairness. Several approaches present formal definitions for livelocks and starvations [33, 39]. A *livelock* occurs when a continuous sequence of interactions amongst a subset of threads prevents other non-deadlocked threads from making progress in completing their tasks. We define a livelock/starvation as a failure to meet a leads-to property $P \rightsquigarrow Q$ under no fairness assumptions. Our motivation for assuming no fairness is multifold. First, we would like to develop correction algorithms that are independent from a specific fairness assumption so the programs they synthesize are portable. Second, correction under no fairness generates programs that will work correctly even if the underlying scheduler fails to generate a fair schedule of execution for threads. Third, while in the literature strong fairness is assumed to simplify the design of livelock/starvation-free concurrent systems, designing strong schedulers is hard and in some cases impossible [34], where a strong scheduler guarantees to infinitely often execute each thread that is enabled infinitely often. As such, to provide portability and resilience to faulty schedulers, we make no assumption on the fairness policy implemented by the thread scheduler.

Formally, a livelock for $P \rightsquigarrow Q$ occurs in a program computation $\sigma = \langle s_0, s_1, \cdots \rangle$ iff P holds in some state s_k, where $0 \leq k$, and Q is false in states $s_k, s_{k+1}, \cdots$. We consider only infinite computations when correcting livelocks. Since synchronization skeletons are *finite state* machines, some states must be repeated in σ. Without loss of generality, we assume that σ includes the sequence $\langle s_k, \cdots, s_i, s_{i+1}, \cdots, s_j, s_i, \cdots \rangle$. The states $s_i, s_{i+1}, \cdots, s_j, s_i$ form a cycle that σ may never leave; i.e., *non-progress cycle*. For example, a thread T_i ($0 \leq i \leq \mathsf{THREADS}-1$) in the swapping program may repeatedly swap its element with another element s, where $s \neq i$, thereby preventing thread T_s to make progress.

3 Problem Statement

In this section, we formulate the problem of finding SSs for safety and leads-to properties. Consider a new property $\mathcal{P}$ that could be either a safety property representing data race-freedom or a leads-to property representing progress (i.e., deadlock/livelock-freedom). Let $p = \langle V_p, \delta_p, I_p \rangle$ be a program that satisfies its specification *spec* from I_p, but does not satisfy $\mathcal{P}$ from I_p. We wish to generate a revised version of p, denoted $p_c = \langle V_p, \delta_c, I_c \rangle$, within the same state space such that p_c satisfies (*spec* $\wedge\, \mathcal{P}$) from I_c. We want to reuse the correctness of p with respect to *spec* by identifying a SS of computations of p generated from I_p. As such, we require that $I_c = I_p$. Moreover, starting in I_c, the transitions of p_c, denoted δ_c, should not include new transitions. Otherwise, p_c may exhibit new computations that do not belong to the computations of p, making it difficult to reuse the correctness of p. Thus, we require that $\delta_c \subseteq \delta_p$. Therefore, we state the problem of finding SSs for correction as follows:

Problem 3.1: Identifying Satisfying Subsets.
Given $p = \langle V_p, \delta_p, I_p \rangle$, its specification *spec*, and a new (safety/leads-to) property $\mathcal{P}$, identify $p_c = \langle V_p, \delta_c, I_c \rangle$ such that:

1. $I_c = I_p$,
2. $\delta_c \subseteq \delta_p$, where $\delta_c \neq \emptyset$, and
3. p_c satisfies (*spec* $\wedge\, \mathcal{P}$) from I_c. □

Previous work [21] illustrates that this problem can be solved in polynomial time in the size of $\mathcal{S}_p$ (1) if we have a *high atomicity* model, where each thread can read all synchronization variables in an atomic step, and (2) if the correction takes place for only one leads-to property. However, finding SSs is known to be NP-hard [21, 13] (in $\mathcal{S}_p$) if a program should be corrected either for multiple leads-to properties, or for one leads-to property where each thread may be unable to read the local state of other threads (i.e., threads have *read restrictions* or *limited observability*).

In this paper, we consider a high atomicity model for several reasons. First, since this work is our first step towards algorithmic correction of concurrency failures, we start with a simpler problem. Second, for many programs, a high atomicity model is a realistic model as synchronization variables (e.g., locks) are often declared as shared variables readable for all threads. Third, we use the failure of correction in the high atomicity model as an *impossibility test*; i.e., the failure of correction in the high atomicity model indicates the failure of correction under read restrictions as well.

4 Extracting Synchronization Skeletons

In this section, we present three of the commonly used synchronization primitives in UPC and present the rules for generating Promela code for them while preserving their semantics based on UPC specification [22]. For the extraction of the synchronization skeleton of Java programs, we reuse the transformation rules available in JavaPathFinder [29].

The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications. The semantics of Promela is based on an operational model that defines how the actions of processes are interleaved. An action (a.k.a *guarded command*) is of the form $grd \rightarrow stmt$, where grd is an expression in terms of program variables and $stmt$ updates program variables. Next, we explain the synchronization constructs in UPC which are used to manage threads interleaving.

4.1 Barriers

Semantics. All threads of a barrier must arrive at the barrier before any of them can proceed.
Syntax. upc_barrier();
Transformation Rule 1.

```
barrier_counter=THREADS; // number of threads
...
barrier_counter=barrier_counter-1;
barrier_counter==0;
barrier_counter=THREADS;
...
```

Notice that a condition in Promela blocks a thread until it is satisfied.

4.2 Split-phase Barriers

Semantics. Split-phase barrier has been devised to reduce the impact of synchronization. The idea is that when a thread completes its global phase of computation, instead of waiting for other threads it notifies other threads of its completion by upc_notify(). Then it starts local computations. When the thread finishes the local processing, it waits for other threads by upc_wait() to enter the next phase of computation.
Syntax.
upc_notify();
upc_wait();
Transformation Rule 2. upc_notify();

```
barrier_counter=THREADS; // number of threads
proceed=0;
...
/* upc_notify */
```

```
barrier_counter=barrier_counter-1;
...     /* local computations */}
```

Transformation Rule 3. upc_ wait();

```
/* upc_wait  */
(barrier_counter==0||proceed==1)->atomic{proceed=1;}
barrier_counter=barrier_counter+1;
(barrier_counter==5||proceed==0)->atomic{proceed=0;}
... /* Next phase of computation */
```

4.3 Locks

UPC uses upc_lock_t as the lock data type. The following functions manipulate locks.

- upc_lock(upc_lock_t *ptr): locks a shared variable of type upc_lock_t.

  Transformation Rule 4.

  ```
  1 bool lk;  // Lock variable
  2 atomic{(lk== ture) -> lk=false; }
  ```

  Line 2 represents an atomic guarded command in Promela that sets the lock variable lk to false (i.e., acquires lk) if lk is available. Otherwise, the atomic guarded command is blocked.

- upc_unlock(upc_lock_t *ptr):unlocks a shared variable of type upc_lock_t.

  Transformation Rule 5.

  ```
  lk = true
  ```

- upc_lock_attemp(upc_lock_t *ptr) tries to lock a shared variable of type upc_lock_t,. If the lock is not held by another thread, it gets lock and returns 1 otherwise it returns 0.

  Transformation Rule 6.

  ```
  atomic{
  /* result denotes the return value.*/
   if
   :: (lk == true) -> lk= false;result=1;
   :: else->result=0;
   fi;}
  ```

Listing 1 illustrates the synchronization skeleton of the swapping program in Promela. Notice that we have added the labels TS and CS to respectively denote the trying section and the critical section of threads. We use these labels to detect whether or not a specific thread is in its trying/critical section. The variable _pid returns a unique integer as the thread identifier.

```
A[_pid] = _pid;
i = _pid;
do
// Randomly generate an index to swap with
if :: s = 0; :: s = 1;
   :: ...     :: s = THREADS - 1;     fi;
TS: {  // Acquire the necessary locks
atomic{(lk[i] == true) -> lk[i] = false;}
atomic{(lk[s] == true) -> lk[s] := false;}
        }
CS: {     . . .  }
// Unlock
lk[s] := true;
lk[i] := true;
od // End of loop
```

Listing 1: Synchronization skeleton of the swapping program of Figure 2 in Promela.

5 Correction of Non-Progress

In our previous work [21, 12], we present a polynomial-time algorithm, called Correct_NonProgress (see Figure 3), for solving Problem 3.1, where the property $\mathcal{P}$ is a leads-to property $P \rightsquigarrow Q$. We demonstrate an intuitive explanation of Correct_NonProgress since we present a new version of this algorithm in Section 5.1. Sections 5.2 and 5.3 respectively present the correction of non-progress for the swapping program and for a classic example in Java.

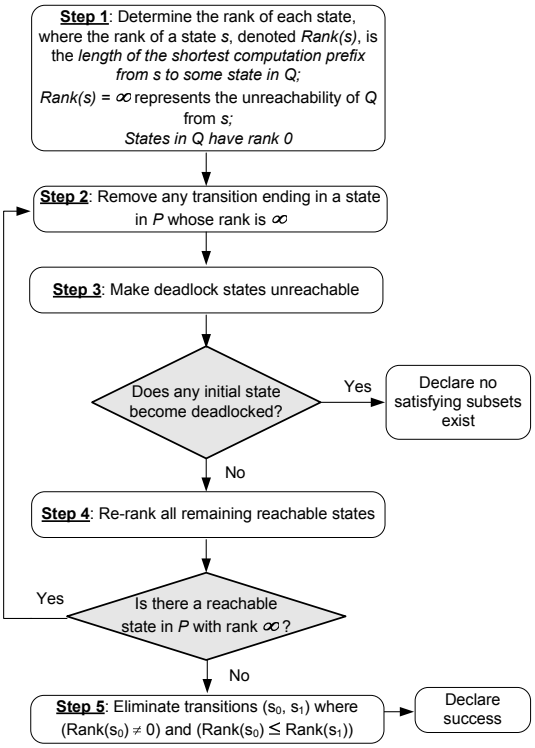

Figure 3: Correct_NonProgress algorithm presented in [12].

In the first step of Figure 3, we compute a rank for each reachable state s, where $Rank(s)$ is the length of the shortest computation prefix from s to some state in Q. The rank of a state in Q is zero and the rank of a state from where there are no computation prefixes to Q is infinity. If there is a transition reaching a state s in P, where $Rank(s) = \infty$, then that transition must be eliminated because P has become true but there is no way Q can be reached. The removal of such transitions may make some states unreachable or create reachable states without any outgoing transitions; i.e., *deadlock* states. For this reason, in Step 3, we resolve deadlock states by making them unreachable. If an initial state becomes deadlocked, then no satisfying subsets exist that satisfy $P \rightsquigarrow Q$ while preserving *spec*. Otherwise, the ranks of remaining states should be recomputed because some states in Q might have been eliminated due to deadlock resolution. If there are still states in P with a rank of infinity, then the above steps should be repeated. Otherwise, a SS exists and we exit the loop in Figure 3. In Step 5, we eliminate all transitions (s_0, s_1) in which $Rank(s_0)$ is non-zero and finite, and $Rank(s_0) \leq Rank(s_1)$. Such transitions participate in *non-progress* cycles. Note that the elimination of such transitions does not create deadlocks because there is at least one computation prefix from s_0 to some states in Q.

5.1 Improved Algorithm

In this section, we present a new version of the algorithm of Figure 3. First, we make two observations about Correct_NonProgress.

Observation 5.1. After Step 2, deadlock states are only in rank 0 and rank ∞.

Proof. Let s be a state such that Rank(s) > 0 and Rank(s) $\neq \infty$. Thus, there is a computation prefix originated at s. Therefore, a finitely-ranked state with a non-zero rank cannot be deadlocked; i.e., the rank of a deadlock state must be either 0 or ∞. □

Observation 5.2. The main loop in the algorithm always terminates in the first iteration. That is, the algorithm never jumps back to Step 2.

Proof. If the algorithm returns to Step 2, then there must exist some infinity-ranked states in P after Step 4. Such states are either deadlock states or states that have no computation prefix to Q. This is a contradiction with what Steps 2 and 3 accomplish. □

Improvements. Based on Observations 5.1 and 5.2, the algorithm Correct_NonProgress can be revised by eliminating the condition right before Step 5. More importantly, we revise Step 4 by replacing the re-ranking of the entire set of reachable states with Algorithm 1 that only updates the ranks to the extent necessary.

Algorithm 1: updateRank(s: state, p: program)

Input: program $p =< V_p, \delta_p, I_p >$ with its state space $\mathcal{S}_p$ and a state s
Output: Rank of s and every state from where s is reachable is updated.

1 $newRank = 1 + Min\{Rank(s_1) | \exists (s, s_1) : (s, s_1) \in \delta_p\}$;
2 **if** $newRank \neq Rank(s)$ **then**
3 | $Rank(s) = newRank$;
4 | For every s_0 for which there is a transition $(s_0, s) \in \delta_p$ updateRank(s_0, p);
5 **end**
6 **return**;

The updateRank algorithm is invoked for every state s if s is not deadlocked, $Rank(s) \neq \infty$ and has an outgoing transition (s, s_d) that reaches a state s_d that has become deadlocked. To update the rank of s, updateRank computes a newRank value that is the minimum of the ranks of the immediate successor states of s (except s_d) plus 1. If newRank $\neq Rank(s)$, then $Rank(s)$ is set to newRank, and the rank of any immediate predecessor state of s is recursively updated. Section 8 illustrates the positive impact of these improvements in the correction time.

Lemma 5.3 The first invocation of updateRank(s,p) , results in the correct rank for s.

Proof. Line 1 of Algorithm 1 guarantees the correct update of Rank(s). □

Lemma 5.4 updateRank(s,p) terminates in polynomial time (in program state space).

Proof. Let G_s be the finite set of states including s and states s_i from where s can be reached by program computation.

Induction base. $G_s = \{s\}$. In this case s has no immediate predecessor. Thus, based on Lemma 5.3, Rank(s) is correctly updated. Since s has no immediate predecessor the algorithm terminates

Induction Hypothesis. Assume the algorithm terminates for any s when $G_s = \{s, s_1, s_2, ..., s_k\}$, we show it will terminate when $G'_s = \{s, s_1, s_2, ..., s_k, s_{k+1}\}$. We have two cases:

Case 1: Invocation of updateRank(s,p) does not cause invocation of updateRank(s_{k+1},p) . In this case, we do not have any extra function call comparing to case $G_s = \{s, s_1, s_2, ..., s_k\}$. Thus, based on our assumption, the algorithm terminates.

Case 2: Invocation of updateRank(s,p) causes invocation of updateRank(s_{k+1},p) . In this case, if Rank(s_{k+1}) does not change, then it terminates. Otherwise, the value Rank(s_{k+1}) is adjusted to its correct value. The new value is propagated back to all its predecessors which is at most $G'_s - \{s_{k+1}\} = G_s$. Thus, based on our assumption, the algorithm will eventually terminate in this case.

To prove that the time complexity of algorithm is polynomial, intuitively observe that the maximum number of recursive calls is the number of transitions in G_s which is less than $|G_s|^2$ where G_s is number of states in G_s. □

Theorem 5.5 updateRank(s,p) returns correct ranks regarding to change in s.

Proof. Based on Lemma 5.3, the first invocation revises Rank(s) correctly. Consider an arbitrary immediate predecessor of s on a shortest computation prefix σ_0 originated from s_0 that reaches Q.

Now, exactly one of the following cases can be true.

Case 1: σ_0 includes s. The invocation of updateRank(s_0,p) on Line 4 causes a recalculation of Rank(s_0) on Line 1. That is, the minimum amongst the ranks of immediate successors of s_0, including recently updated

s, results in updating Rank(s_0) in Line 3 because the shortest computation prefix that determined the rank of s_0 passes through s. Therefore, rank of s_0 is correctly updated.

Case 2: σ_0 does not include s. Since the shortest computation prefix reaching Q (from s_0) does not include s, and the update on the rank of s cannot decrease Rank(s), Lines 3 and 4 are not executed due to the invocation updateRank(s_0,p) . Therefore, rank of s_0 (and accordingly rank of any state from where s_0 can be reached) remains correctly unchanged. □

5.2 Case Study: Swapping Program

In this section, we use the swapping program as a running example to demonstrate how the improved Correct_NonProgress ensures that if each one of the threads tries to swap two elements, then it will eventually get a chance to do so irrespective of fairness policy.

Model creation. While we have implemented our correction algorithms using both an explicit-state method and BDDs [14], for illustration purposes, we represent synchronization skeletons as reachability graphs in this section. A Reachability Graph (RG) is a directed graph in which nodes represent synchronization states and arcs denote atomic transitions. Figure 4 illustrates the deadlock-free RG of the synchronization skeleton of a version of the swapping program with two threads; i.e., THREADS= 2. Each oval represents a state in which only the values of the predicates TS_i and CS_i $(i = 0, 1)$ are illustrated. We have generated the deadlock-free RG by making deadlock states unreachable (see Step 3 of Figure 3). For brevity, we have abstracted out the values of other variables in Figure 4. For this reason, some states appear as duplicates. We denote the initial state by a rectangle. The labels on the transitions denote the index of the thread that executes that transition.

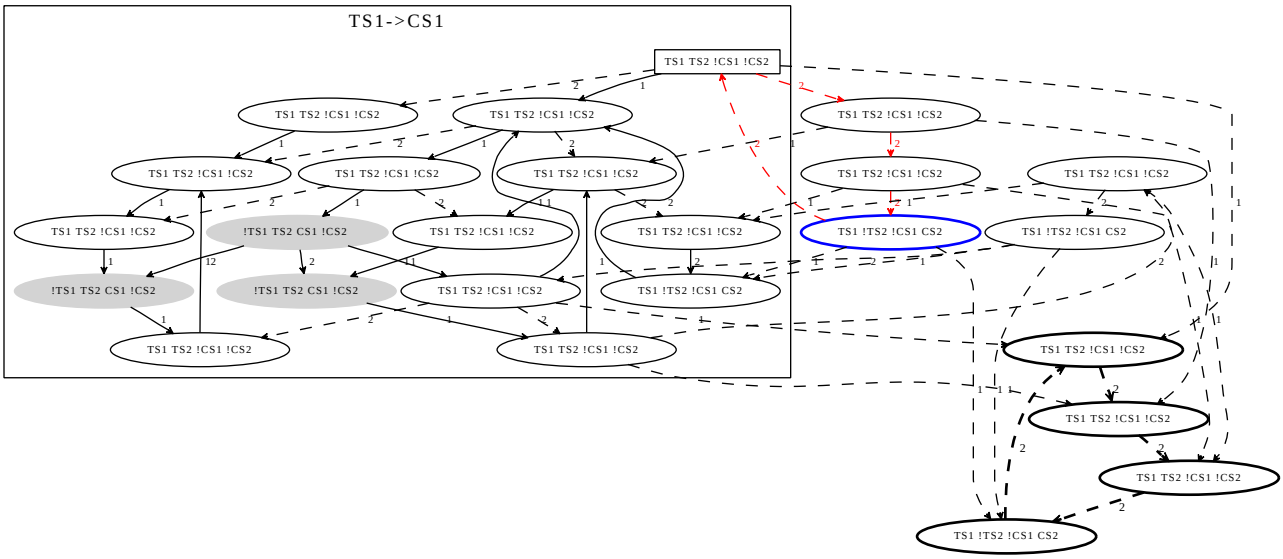

Figure 4: The reachability graph of the synchronization skeleton of the UPC program in Figure 2.

Step 1: Ranking. Since we first correct the swapping program for the property $TS_1 \rightsquigarrow CS_1$, we rank each state s based on the length of the shortest computation prefix from s to some state where CS_1 holds (shaded states in Figure 4). Figure 5 illustrates a ranked RG of the boxed part of Figure 4.

Step 2: Making infinity-ranked states unreachable. The **bold** states on the lower right corner of Figure 4 constitute the set of states in TS_1 from where there are no computation prefixes reaching CS_1; i.e., infinity-ranked states. In this step, we eliminate any transition reaching infinity-ranked states.

Step 3: Making deadlock states unreachable. Making infinity-ranked states unreachable does not introduce deadlock states for the swapping program.

Step 4: Update ranks. Since no deadlock states were created in the previous states, the ranks of the remaining states are not updated.

Step 5: Eliminating rank-violating transitions. The non-progress cycles causing the failure of $TS_1 \rightsquigarrow CS_1$ are contained in the set of dashed transitions that start in a non-zero rank and terminate in either the same rank, or a higher rank. For example, let (s_0, s_1) denote the outgoing dashed red transition from the initial state in Figure 4. The rank of the initial state s_0 is 3 and the rank of s_1 is also 3. The last step of the algorithm in Figure 3 eliminates (s_0, s_1), thereby making all states in the non-progress cycle illustrated with the Red unreachable. (Notice that the dashed blue transitions in Figure 5 are preserved because they start in Rank 0.) As such, the states outside the box in Figure 4 are no longer reachable in the revised RG of Figure 5. The solid arrows in Figure 5 illustrate a SS that satisfies $TS_1 \rightsquigarrow CS_1$.

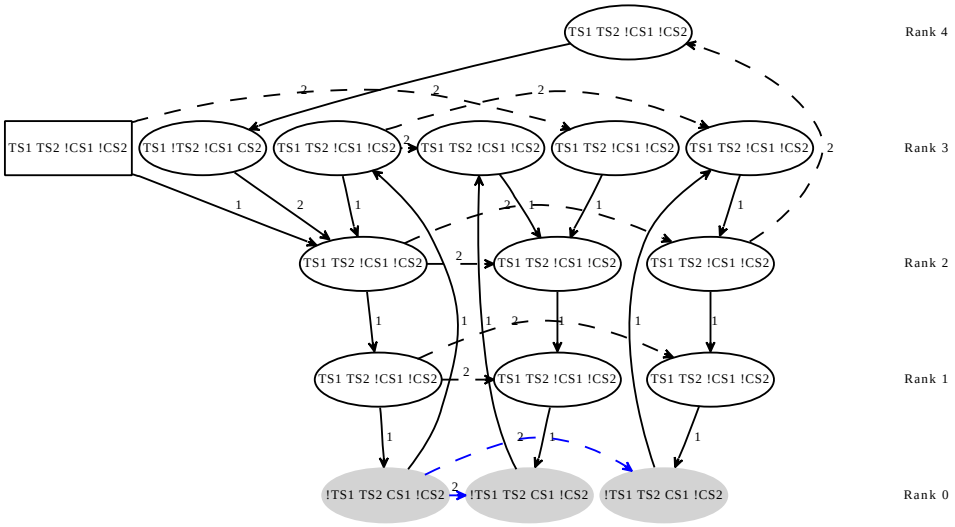

Figure 5: The revised ranked reachability graph for $TS_1 \rightsquigarrow CS_1$ derived from Figure 4.

Our ranking algorithm is conservative as there may be a valid SS that includes some of the transitions ruled out by our approach. For example, eliminating the red transition reaching the blue state in Figure 4 instead of the red transition originating at the initial state would resolve the red non-progress cycle without making its states unreachable; i.e., would create a SS with more states/transitions. Ideally, we wish to eliminate the least number of states/transitions so we can identify a *maximal* SS. Finding a maximal SS would increase the chances of success for correcting other non-progress failures. While we have illustrated that, in general, finding a maximal SS is a hard problem [12], we are currently investigating how we can reduce the complexity of identifying maximal SSs using sound heuristics and sufficient conditions.

We apply Correct_NonProgress to the intermediate program resulting from the correction of non-progress for $TS_1 \rightsquigarrow CS_1$ in order to correct non-progress failures for $TS_2 \rightsquigarrow CS_2$. The code in Listing 4 Appendix A represents the final corrected UPC code.

The revised code utilizes two shared arrays s and setS to make local state of a thread observable to other threads. Shared variable setS[i] represents the state of thread i. Value setS[i]=0 denotes that thread i is not in its try section (i.e. $\neg TS_i$). Otherwise, thread i either is trying to enter the critical section, or it is already in the critical section. We use upc_mem_get() function to access shared values, because the UPC run-time system does not guarantee to return the latest values of a shared variable located in another thread's affinity. In the program, SS1 and SS2 keep local copies of setS[1] and setS[2] . Variable s[i] denotes the target element to swap with thread i. Line 27 shows that thread 1 swaps its shared data with thread 2 and respectively Line 46 shows that thread 2 swaps its shared data with thread 1.

Line 26 blocks execution of thread 1 when thread 2 is in its trying section. The initial value s[1] =0 allows thread 1 to enter its trying section first. In addition, Line 45 of thread 2, gives the priority to thread 1 to try the critical section and enter it. Respectively, while loop, in Line 32 in thread 1, ensures thread 2 to proceed to its trying section. This way, when thread 1 exits the critical section (Line 35), thread 2 enters the critical

section and vice-versa. Thus, both threads satisfy their progress properties.

We use the following transformation rule to translate the busy waiting conditions in Promela to the corresponding UPC statements.

Syntax: shareVar==condition

Semantics: It blocks execution of the thread until shareVar is equal to condition.

Transformation Rule 7.

```
int localCopy;  /* local copy of a shared variable sharedVar_i */
upc_memget(&localCopy, &sharedVar[i], sizeof());
while(localCopy!=condition)  upc_memget(&localCopy, &sharedVar[i],sizeof());
```

5.3 Single Lane Bridge

In this section, we illustrate the algorithmic correction of non-progress failures in a concurrency example adapted from [35]. Consider a bridge, modeled by a Java class Bridge, over a river that has only one lane, i.e. at any time cars can move in only one direction; otherwise a crash could occur. There are two types of cars, namely the *Red* and the *Blue* cars; red cars move from right to the left on the bridge, and blue cars move in the opposite direction. The Single Lane Bridge (SLB) program includes two threads, namely RedCar and BlueCar, that model the cars. The Bridge class has the following state variables: an integer variable nred that captures the number of red cars on the bridge, a variable nblue that denotes the number of blue cars on the bridge, a variable capacity that represents the capacity of the bridge, a variable onbridge representing the total number of cars on the bridge, and a random variable choice that represents the decision of a car for leaving or staying on the bridge. The RedCar thread (respectively, BlueCar) continuously invokes the redEnter method (Lines 1-5 in Figure 6) and redExit method in Lines 6-9 in Figure 6 (respectively, blueEnter and blueExit methods). Listing 6 in Appendix B shows the synchronization skeleton of program in Figure 6.

```
1 synchronized void redEnter()
2       throws InterruptedException {
3 if ((capacity>onbridge) ) {
4                 ++nred;  onbridge++; }
5 }
6 synchronized void redExit(){
7  if(nred>0)\{ choice=generator.nextInt(2);
8              if(choice==1) --nred;  }
9  }
10 synchronized void blueEnter()
11       throws InterruptedException {
12 if ((capacity>onbridge)) {
13                  ++nblue;  onbridge++; }
14  }
15 synchronized void blueExit() {
16  if(nblue>0){ choice=generator.nextInt(2);
17               if(choice==1) --nblue;  }
```

Figure 6: The methods of the Bridge class SLB program.

Correction for ensuring safety. To enforce safety to the program, the values of nred and nblue must not be more than 0 at same time. Thus we applied $\Box!(nred > 0 \wedge nblue > 0)$ as the safety property. The result is shown in Figure 7. The bold texts in this figure show the changes to the original program to make it safe. This safety property can be assumed as a mutual exclusion problem.

While the SLB program guarantees mutual exclusion; i.e., no crashes occur on the bridge, one thread may take over the bridge by continuously sending cars to the bridge, thereby depriving the other type of cars to cross the bridge. To address this non-progress failure, we use the improved Correct_NonProgress algorithm to ensure that the leads-to properties $true \rightsquigarrow (nblue > 0)$ and $true \rightsquigarrow (nred > 0)$ hold respectively for the BlueCar and RedCar threads.

Correction for $true \rightsquigarrow (nred > 0)$**.** In this step, the resulting program blocks thread BlueCar from using the bridge, thus guaranteeing progress for RedCar thread.

```
1  synchronized void redEnter()
2        throws InterruptedException {
3  if ((capacity>onbridge)  && (nblue==0) ) {
4                  ++nred;  onbridge++; }
5  }
6  synchronized void redExit(){
7   if(nred>0){ choice=generator.nextInt(2);
8               if(choice==1) --nred;  }
9   }
10 synchronized void blueEnter()
11       throws InterruptedException {
12 if ((capacity>onbridge)  && (nred==0) ) {
13                  ++nblue;  onbridge++; }
14  }
15 synchronized void blueExit() {
16  if(nblue>0){ choice=generator.nextInt(2);
17               if(choice==1) --nblue;  }
18  }
```

Figure 7: The methods of the Bridge class SLB program after correction of safety violation.

Correction for $true \rightsquigarrow (nblue > 0)$**.** The correction of the resulting program from the previous step fails for $true \rightsquigarrow (nblue > 0)$.

There are two main causes for the failure of correcting multiple non-progress failures. First, for some programs and leads-to properties, the order of correcting different non-progress failures may impact the success of correction because some transitions eliminated for the correction of a non-progress failure might be useful for the correction of other failures. Second, for some programs and leads-to properties, there may not exist a SS in program state space that meets the requirements of all progress properties the program is supposed to satisfy, no matter which order of correction we select. This is the case for the SLB program. Next, we explain a heuristic we use to address this problem by expanding the state space of SLB and adding computational redundancy.

Heuristic for finding SSs in an expanded state space. One cause of the failure of Correct_NonProgress in correcting multiple non-progress failures is the lack of sufficient computational redundancy and non-determinism in the faulty program. In fact, while non-determinism may be a cause of concurrency failures, the chance of successful correction of a concurrency failure depends on the degree of non-determinism available in the faulty program; the more non-determinism the higher chance of successful correction! With this intuition, we present the following heuristic for cases where correction of multiple non-progress failures fails.

First, we manually add new variables and/or instructions to the failed program based on the principle of *computation superimposition* proposed in [16]. Then, we algorithmically correct the failures in the superimposed program. The new variables and instructions just monitor the computations of the faulty program without interfering with them. In the case of SLB, we add a new shared variable, denoted procStamp, to the Bridge class. The procStamp variable is used to keep a record of the last thread that entered its critical section. Each thread is assigned a unique stamp; the stamp of RedCar is 1 and the stamp of BlueCar is 2. The superimposed code appears in Lines 5 and 6 below.

```
1  synchronized void redEnter()
2        throws InterruptedException {
3  if ((capacity>onbridge)  && (nblue==0) ) {
4                  ++nred;  onbridge++;
5            if(procStamp==1)  procStamp=3;
6            else              procStamp=2;      }
7  }
```

If RedCar (respectively, BlueCar) enters its critical section and procStamp is equal to 1 (respectively, 2), then it sets procStamp to 3 illustrating that RedCar (respectively, BlueCar) just made progress by entering its critical section. Otherwise, RedCar (respectively, BlueCar) gives priority to BlueCar (respectively, RedCar) by setting procStamp to 2 (respectively, 1). Note that the superimposed SLB program still has the non-progress failures.

Theorem 5.6. The corrected SLB program satisfies $(true \rightsquigarrow (\mathsf{procStamp} = 3))$ *iff* the corrected SLB program satisfies $true \rightsquigarrow (nred > 0)$ and $true \rightsquigarrow (nblue > 0)$.

Proof.

$\Rightarrow$. If $true \rightsquigarrow$ (procStamp $= 3$) holds then predicates $true \rightsquigarrow (nred > 0)$ and $true \rightsquigarrow (nblue > 0)$ are satisfied. By contradiction, assume one of the predicates does not hold; e.g., $true \rightsquigarrow (nblue > 0)$. Thus, RedCar continuously uses the bridge while BlueCar starves. In this case, infinite invocation of redEnter implies that procStamp remains 2, which contradicts $true \rightsquigarrow$ (procStamp $= 3$). The same reasoning is correct when $true \rightsquigarrow (nblue > 0)$ does not hold. Therefore, $\Rightarrow$ part holds.

$\Leftarrow$. $true \rightsquigarrow (nBlue > 0)$ and $true \rightsquigarrow (nRed > 0)$ hold, then $true \rightsquigarrow$ (procStamp $= 3$) holds. By contradiction, assume $true \rightsquigarrow$ (procStamp $= 3$) does not hold. Since procStamp $= 3$ denotes the resource alternation, at least one of threads should fail to progress and it contradicts the assumption that both threads proceed using the bridge. Therefore, the the proof follows. □

Correction of the superimposed SLB for ($true \rightsquigarrow$ (procStamp $= 3$)). To ensure that both threads RedCar and BlueCar make progress, we use Correct_NonProgress to correct the superimposed SLB program for the leads-to property ($true \rightsquigarrow$ (procStamp $= 3$)). First, the condition of the 'if statement' on Line 3 in Figure 6 is replaced with the condition (onbridge==0); i.e., RedCar can enter the bridge only when there are no cars. Second, the condition on the choice variable in Line 8 is eliminated. That is, a red car that invokes the redExit has no choice of deciding whether to leave the bridge or not; it has to leave the bridge. Third, a busy waiting is synthesized in the synchronization skeleton of the RedCar to ensure that a blue car can use the bridge before another red car does so. The translation of this busy waiting is the following method added to the Bridge class.

```
1 synchronized void redWait(){
2         while(procStamp!=1)  wait();      }
3 synchronized void blueWait(){
4         while(procStamp!=2)  wait();      }
```

Moreover, the corrected RedCar thread continuously invokes redEnter, redExit and redWait methods instead of just redEnter and redExit. (Symmetric revisions occur for the BlueCar thread.) Listings 5- 8 in Appendix B shows the complete body of Bridge, RedCar and BlueCar.

6 Correction of Data Races

In this section, we discuss the correction of data races in the context of two case studies. Starting from initial states, the Correct_Safety algorithm presented in [21] first eliminates any transition that reaches a bad state; i.e., a state not in S for a safety property $\Box S$. Such removal of transitions may create some deadlock states. Subsequently, Correct_Safety ensures that no deadlock state is reached by the computations of p from I_p. If an initial state becomes deadlocked, then the algorithm declares failure in correcting the program. Otherwise, a corrected program is returned.

Our contribution in this section is that we design and implement a version of Correct_Safety for algorithmic correction of classic data races.

6.1 Case Study: Heat Flow

The Heat Flow (HF) program includes THREADS>1 threads and a shared array t of size THREADS$\times regLen$, where $regLen > 1$ is the length of a region vector accessible to each thread. That is, each thread i ($0 \leq i \leq$ THREADS-1) has read/write access to array cells $t[i * regLen]$ up to $t[((i+1) * regLen) - 1]$. The shared array t captures the transfer of heat in a metal rod and the HF program models the heat flow in the rod. We present an excerpt of the UPC code of HF as follows:

```
1 shared double t[regLen*THREADS];
2 double tmp;
3
4 base = MYTHREAD*regLen;
5 for (;;) {
6  // Perform some local computations
7  for (i=base+1; i<base+regLen-1; ++i) {
8    tmp = (t[i-1] + t[i] + t[i+1]) / 3.0;
9    // Perform some local computations
10   t[i-1] =  // Assign the results to t[i-1]
11   }
```

```
12  if ( MYTHREAD < THREADS-1) {
13    tmp = (t[base+regLen-2] +
14           t[base+regLen-1] +
15     t[base+regLen]) / 3.0;
16    // Perform some local computations
17    t[base+regLen-1] = tmp;   }
18  upc_barrier;
19  t[base+regLen-2] = // some expression;
20 }
```

Each thread continuously executes the code in Lines 6 to 20. Specifically, in Lines 7 to 11, each thread i, where $0 \leq i \leq \mathsf{THREADS}-1$, first computes the heat intensity of the cells $t[base]$ to $t[base+regLen-3]$ in its own region. Subsequently, every thread, except the last one, updates the heat intensity of $t[base+regLen-1]$ (see Lines 12-18). Before updating $t[base+regLen-2]$ in Line 20, all threads synchronize using upc_barrier. Then they compute an expression that is assigned to $t[base+regLen-2]$. Observe that, before each thread i, for $0 \leq i \leq \mathsf{THREADS}-2$, updates the last value of its region in Line 17, it needs to read the first value of its successor's region in Line 15 (See figure 8). This is the place where a data race might occur between a thread executing Line 10 in the first iteration of the for loop in Line 7 and its predecessor executing Line 15.

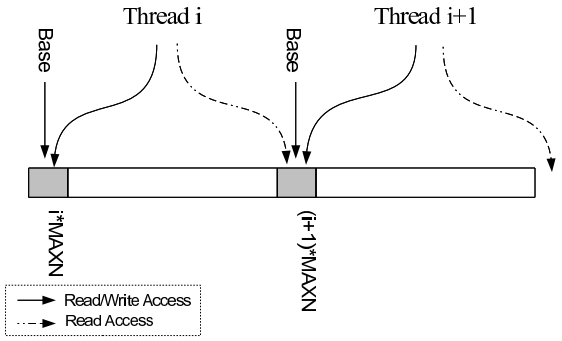

Figure 8: Data Access behavior of threads in heatflow example.

Specifying data races as safety properties. To capture data race-freedom as a safety property, we specify a state predicate that represents an invariance condition. Towards this end, we consider two arrays W and R with THREADS number of cells. Every time thread T_i writes $t[base]$ in Line 10, it increments $\mathsf{W}[i]$, and when the predecessor of T_i wants to read $t[base]$ it increments $\mathsf{R}[i]$. Since in this case only one thread has access right to each one of $\mathsf{W}[i]$ and $\mathsf{R}[i]$, no race conditions occur on these shared data. Nonetheless, in general, we perform these actions atomically in Promela to ensure that $\mathsf{W}[i]$ and $\mathsf{R}[i]$ do not become a point of contention themselves.[3] For the HF program, ensuring that the safety property $\Box(((W[i]=0) \wedge (R[i] \geq 0)) \vee ((W[i]=1) \wedge (R[i]=0)))$ holds for $0 \leq i \leq THREADS-1$ guarantees data race-freedom.

Nonetheless, for illustration, we present the translation of Lines 10 and 15 in Promela as follows:

```
1 int W[THREADS]=0;
2 int R[THREADS]=0;
3 active [THREADS] proctype heatFlow (){
4     .   .   .
5// Lines 6 and 7 correspond to Line 10 of HF
6 atomic{ W[_pid]=W[_pid]+1; }
7 atomic{ W[_pid]=W[_pid]-1; }
8     .   .   .
9// Lines 10 and 11 correspond to Line 15 of HF
10 atomic{R[_pid+1]=R[_pid+1]+1; }
11 atomic{R[_pid+1]=R[_pid+1]-1; }
12     .   .   .
13}
```

[3]Most modern programming languages support test-and-set operations on single variables (e.g., in Berkeley UPC, a family of upc_atomic() functions, and in Java, Atomic Variables).

Corrected Heat Flow. The corrected program uses a lock variable $lk[i]$ for each thread T_i ($1 \leq i \leq THREADS - 1$) and its predecessor. Specifically, in the code of HF, before T_i executes Line 10, it must acquire $lk[i]$. Likewise, thread T_{i-1} must acquire $lk[i]$ before it reads $t[base + regLen]$ in Line 15.

Caveats. We are currently investigating the problem of generating synchronization skeletons for programs where the pattern of access to shared data is dynamically changed at run-time. For example, consider a more accurate HF program in which each thread needs to read the entire shared array to have a more precise computation of heat intensity for its own region. As a result, depending on which cell of the shared array a thread needs to read/write, a data race may or may not occur. For such applications, correction of data races is more difficult.

6.2 Readers Writers

In this section, we apply our method on a classic example in concurrency. Consider a shared resource and two reader and two writer threads . Each reader (respectively, writer) continuously reads (respectively, writes) the data. The specification of this program has two state variables rdr and wrtr which respectively denote the number of readers and writers at any moment. Listing 2 shows the body of threads in Promela. It is possible that writers intervene the readers and cause a data race. Thus we added a safety property to avoid the race condition.

```
int pc[4]=0;
int rdr=0, wrtr=0;
active [4] proctype readerwriter()
{
do
::(pc[_pid]==0)->
 atomic{
/* Threads 0 and 1 are readers and threads 2 and three are writers*/
 if
 ::(_pid==0||_pid==1)-> pc[_pid]= 1;
 ::else ->pc[_pid]= 5;
 fi;
 }
::(pc[_pid]==1)->atomic {pc[_pid]=2;}
::(pc[_pid]==2)->atomic {
 /* When a reader starts reading shared resource increase value rdr   */
 rdr=rdr+1;pc[_pid]= 3;}
::(pc[_pid]==3)->atomic {
 /* When a reader finishes reading shared resource decreases value rdr   */
 rdr=rdr-1;pc[_pid]= 1;}
::(pc[_pid]==5)->atomic {
 if
 ::(_pid==2||_pid==3)-> pc[_pid]= 6;
 ::else->skip;
 fi}
::(pc[_pid]==6)->atomic {pc[_pid]=7;}
::(pc[_pid]==7)->atomic{
 /* When a writer starts writing shared resource increase value wrtr */
 wrtr=wrtr+1;pc[_pid]= 8;  }
::(pc[_pid]==8)->atomic{
 /* When a writer finishes writing shared resource it decreases value wrtr */
 wrtr=wrtr-1;pc[_pid]= 6;}
od
}
```

Listing 2: Readers Writers Specifications in Promela .

Correction for ensuring safety. The safety property is that writers should not intervene the readers, in addition the writers should not write shared data simultaneously. Thus we added $\Box(rdr \geq 0 \wedge wrtr == 0) \vee (rdr = 0 \wedge wrtr = 1)$. Listing 3 shows the safe reader writes program. For readers, it enforces that at most one writer can write on the shared resource and respectively a writer can write only if no reader is reading and no writer is writing.

```
int pc[4]=0;
int rdr=0, wrtr=0;
active proctype reader0()
{
do
::(pc[0]==0)->{ pc[0]= 1;}
::(pc[0]==1)->atomic {pc[0]=2;}
::(pc[0]==2)->atomic {
 (wrtr==0)/* This condition prohibits interference of writers*/
 ->rdr=rdr+1;pc[0]= 3;}
::(pc[0]==3)->atomic {rdr=rdr-1;pc[0]= 1;}
od
}
active proctype reader1()
{
do
::(pc[1]==0)->{ pc[1]= 1;}
::(pc[1]==1)->atomic {pc[1]=2;}
::(pc[1]==2)->atomic {
   (wrtr==0)/*This condition prohibits interference of writers*/
   ->rdr=rdr+1;pc[1]= 3;}
::(pc[1]==3)->atomic {rdr=rdr-1;pc[1]= 1;}
od
}

active proctype writer0()
{
do
::(pc[2]==0)->{pc[2]= 6;}
::(pc[2]==6)->atomic {pc[2]=7;}
::(pc[2]==7)->atomic{
  (wrtr==0&&rdr==0)/*This condition prohibits interference with readers
       or other writer*/
  ->wrtr=wrtr+1;
    pc[2]= 8;}
::(pc[2]==8)->atomic{wrtr=wrtr-1;pc[2]= 6;}
od
}

active proctype writer1()
{
do
::(pc[3]==0)->{pc[3]= 6;}
::(pc[3]==6)->atomic {pc[3]=7;}
::(pc[3]==7)->atomic{
  (wrtr==0&&rdr==0)/*This condition prohibits interference with readers
       or other writer*/
  ->wrtr=wrtr+1;pc[3]= 8;}
::(pc[3]==8)->atomic{wrtr=wrtr-1;pc[3]= 6;}
od
}
```

Listing 3: Readers Writers specifications in Promela after correction for ensuring safety.

7 Other Case Studies

In this section we demonstrate application of our tool on three other examples. Section 7.1 presents correction for safety and progress in a HPC synchronization construct called “barrier synchronization”. Section 7.2 applies the correction method to impose a progress property in Token Ring protocol. Section 7.3 uses an extension of SLB program called ”Two sinlgle lane bridge” to address the safety violation and non-progress defects in this program.

7.1 Barrier Synchronization

In this section, we use a barrier synchronization to demonstrate do-ability of our approach. BarSync is used in HPC applications for synchronizing the results of threads in round-based computations. Consider three threads where each of them could circularly rotate between the positions ready, execute and success (See Listing 9 in Appendix C for detailed description.) Variable pc1, pc2 and pc3 denote the positions where threads are in. The invariant of the Barrier specifies that at least two threads should be in the same position.

Correction for ensuring safety. To enforce the invariant, we add the following property to the program in Listing 9.

$(\forall j : 1 \leq j \leq 3 : (pc[j] \neq ready)) \vee (\forall j : 1 \leq j \leq 3 : (pc[j] \neq execute)) \vee (\forall j : 1 \leq j \leq 3 : (pc[j] \neq success))$ Listing 10 in Appendix C is the result of this correction.

Correction for ensuring progress. The program statement requires that, starting from the state $allR =< ready, ready, ready >$ all threads will eventually synchronize in the state $allS =< success, success, success >$. Listing 11 of Appendix C shows the new program that imposes $allR \leadsto allS$.

7.2 Token Ring

In this section, we demonstrate the application of our tool on a distributed control protocol called Token ring. Token ring protocol is often used for ensuring mutual exclusion, where only the thread that has the token can access shared data, and at all times at most one thread has a token. Assume three threads on a ring. Each of thread k has a variable x*k* to update and they are able to read the variable of its neighbor. Listing 12 in Appendix D shows the specification of this threads. In this case we add the progress property that specifies that infinitely often the threads should have the same value.

Correction for ensuring progress. Program statement requires to add $true \leadsto (x1 = x2 \wedge x2 = x3)$ as the progress property. Listing 13 in Appendix D illustrates the resulting program after this correction.

7.3 Two Single Lane Bridges

In this section, we extend the example in 5.3. In this example we have two threads RedCar and BlueCar as described in 5.3 and two single lane bridge. In this example, each thread randomly decides on a bridge to use and then it puts a car on the selected bridge. br[k] denotes the bridge that thread k is going to use. Listing 14 in Appendix E demonstrates specification of this problem

Correction for ensuring safety. To make the bridge safe, we should guarantee that the both threads do not collide on any bridge i, i.e. $\forall i : \Box\neg(nred_i > 0 \wedge nblue_i > 0)$. Listing 15 in Appendix E shows the specification of two threads after ensuring this property.

Correction for ensuring progress. To add progress to both threads, we sequentially add leads-to property to the threads. We first add $true \leadsto total[0] > 0$ and then we add $true \leadsto total[1] > 0$. Listing 16 in Appendix E shows the outcome of this process.

8 Experimental Results

In this section, we summarize our experimental results. Our objective in this section is to illustrate the feasibility of our algorithmic correction method in practice. Moreover, we demonstrate (in Table 9) that the improved version of Correct_NonProgress presented in Section 5.1 indeed provides an average 27% run-time improvement in our case studies using an explicit-state implementation. We ran the experiments on a machine with a DualCore Intel CPU (3.00 GHz) with 2 GB RAM and the Linux Fedora operating system. We also have conducted a case study on an extended version of the SLB program in Section 5.3, where there are two bridges. We did not observe a significant improvement in run-time for the swapping program because updateRank almost explores 90% of the reachable states for correcting the non-progress failures of the second thread. For token ring, there were no infinity-ranked states. Thus, no deadlock states were created. The improved Correct_NonProgress skips the re-ranking in Step 4 of Figure 3, whereas the original algorithm would re-rank the entire set of reachable states. For this reason, we observe a significant improvement in the case of token ring.

We have corrected non-progress failures of a version of BarSync with 150 threads in half an hour using a symbolic implementation of Correct_NonProgress. However, we could not scale up the correction of the token

	Data Race Correction	Non-Progress Correction	Correction Time of Correct_NonProgress	Correction Time of Improved Correct_NonProgress	Percentage of Run-Time Improvement
Heat Flow (2 threads)	✓	✓	0.503	0.33	34.39
Heat Flow (4 threads)	✓	✓	79.86	65.74	17.68
Readers/Writers (2 readers, 2 writers)	✓	✓	0.61	0.49	19.67
Single Lane Bridge		✓	0.156	0.109	30.13
Swapping		✓	0.57	0.53	7.018
Two Single Lane Bridges		✓	20.735	16.54	20.23
Barrier Synchronization (10 thread)		✓	6680.5	4520	32.34
Token Ring (10 threads)		✓	2.36	0.94	60.17

Figure 9: Experimental results. (Time unit is milliSec.)

ring program beyond 7 threads using the symbolic implementation, whereas with the explicit-state implementation we were able to correct a token ring program with 10 threads! We observe that the space efficiency of BDDs [14] dramatically drops where program variables have large domains. By contrast, symbolic correction better handles programs that have variables with small domains, but a high degree of non-determinism (i.e., large number of threads).

9 Limitations

In this section, we discuss some theoretical and practical limitations of our approach. In principal, we have previously illustrated that finding a SS that meets the requirements of multiple leads-to properties is a hard problem [21]. As such, if we want to have efficient tools for automatic correction of non-progress failures, then we should develop polynomial-time sound (but incomplete) heuristics. If such heuristics succeed in correcting non-progress failures of several leads-to properties, then the programs they generate are correct, however, they may fail to correct programs. Another theoretical limitation of identifying SSs is when threads have read restrictions with respect to the synchronization state of other threads. Under such read restrictions (i.e., limited observability [42]), correcting both safety-violations and non-progress failures become hard problems [13]. To address this challenge, we adopt a two-track approach. In the first track, we develop sound heuristics that reduce the complexity of correction under read restrictions. In the second track, we first correct programs in the high atomicity model (i.e., full observability) to determine how much of the synchronization state of each thread should be exposed to other threads. Then, we leverage the existing synchronization primitives to enable threads in reading the state of other threads. For instance, we are investigating the use of lock variables. Consider a scenario where two threads T_1 and T_2 compete for a lock lk. When T_1 observes that lk is unavailable, T_1 gains the knowledge that T_2 is executing the piece of code protected by lk (e.g., is trying to enter its critical section). In other words, T_1 reads the synchronization state of T_2 by trying to acquire lk. However, the problem is that T_1 gets blocked on lk using traditional locking primitives (such as `upc_lock()` function in UPC). Thus, what we need is to check whether or not lk is

available without waiting for it. Fortunately, UPC has the function upc_lock_attempt() (defined in Section 4) that permits us to detect whether or not another thread is in a specific synchronization state. Likewise, the Lock objects in Java enable us to check the availability of a lock (using a tryLock method) without blocking on it.

A practical caveat is the realization of automatically generated corrections in the target programming language; not all corrections generated in Promela code are realizable in UPC and Java. We categorize the corrections into three sets, namely additional preconditions on existing instructions, locking mechanisms and statements that should be executed atomically. Most modern programming languages provide constructs for realizing the second and the third sets of corrections. For example, for atomic execution of statements, Java Concurrency API provides Atomic types (e.g., AtmoicInteger) for atomic updates of single variables. For the realizability of preconditions, we follow the approach we explained above for dealing with limited observability.

10 Conclusions and Future Work

We presented the concept of *satisfying subsets* that provides a foundation for algorithmic correction of concurrency failures, including non-progress and data races. Given a program that has a (non-progress or data race) failure, we first generate a slice of the program capturing inter-thread synchronization functionalities, called the *synchronization skeleton*. Then we generate a finite model of the synchronization skeleton and systematically search for a subset of program computations that avoid the concurrency failures; i.e., a Satisfying Subset (SS). The existence of a SS results in pruning the unnecessary non-determinism that causes the failure. We have validated our approach in the context of several multithreaded programs in the Unified Parallel C (UPC) programming language and in Java. We are currently extending this work in several directions including (i) the development of sound heuristics for identification of SSs that meet multiple progress requirements; (ii) the parallelization of correction algorithms towards increasing the scalability of our approach, and (iii) algorithmic correction of high-level data races [8, 15] and consistency violation of atomic data sets [40, 28].

References

[1] Berkeley UPC Mailing List. `http://www.nersc.gov/hypermail/upc-users/0437.html`.

[2] Berkeley UPC User's Guide. `http://upc.lbl.gov/docs/user/index.html#bupc_poll`.

[3] Java Virtual Machine. `http://java.sun.com/j2se/1.5.0/docs/guide/vm/thread-priorities.html#general`.

[4] Spin language reference. `http://spinroot.com/spin/Man/promela.html`.

[5] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[6] S. Akhter and J. Roberts. *Multi-Core Programming*. Intel Press, 2006.

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

[8] C. Artho, K. Havelund, and A. Biere. High-level data races. In *The 1st International Workshop on Validation and Verification of Software for Enterprise Information Systems*, pages 82–93, 2003.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[10] C. Barton, C. Carscaval, G. Almasi, Y. Zheng, M. Farrens, and J. Nelson. Shared memory programming for large scale machines. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–117, 2006.

[11] J. Bloch. *Effective Java*. Addison-Wesley, 2 edition, 2008.

[12] B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):2009, 1–28.

[13] B. Bonakdarpour and S. S. Kulkarni. Revising distributed unity programs is np-complete. In *12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.

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

[15] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. *Concurrency - Practice and Experience*, 16(12):1161–1172, 2004.

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

[17] S. Cherem, T.Chilimbi, and S. Gulwani. Inferring locks for atomic sections. *ACM SIGPLAN Notices*, 43(6):304–315, 2008.

[18] L. Chew. A System for Detecting, Preventing and Exposing Atomicity Violations in Multithreaded Programs. Master's thesis, University of Toronto, 2009.

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

[20] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *IEEE/ACM International Conference on Software Engineering*, pages 442–452, 2002.

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

[22] T. A. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *The ACM/IEEE Conference on High Performance Networking and Computing*, page 27, 2006.

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

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

[25] C. Flanagan and S. N. Freund. Automatic synchronization correction. *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[26] C. Flanagan and S. N. Freund. Type inference against races. *Science of Computer Programming*, 64(1):140–165, 2007.

[27] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *ACM SIGPLAN Notices*, 38(5):338–349, 2003.

[28] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering*, pages 231–240, 2008.

[29] K. Havelund. Java PathFinder: A Translator from Java to Promela. In *SPIN Workshops*, page 152, 1999.

[30] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

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

[32] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *International Conference on Computer Aided Verification (CAV)*, pages 226–238, 2005.

[33] Y.-S. Kwong. On the absence of livelocks in parallel programs. In *Proceedings of International symposium on semantics of concurrent computations*, pages 172–190, 1979.

[34] M. Lang and P. A. G. Sivilotti. On the impossibility of maximal scheduling for strong fairness with interleaving. In *International Conference on Distributed Computing Systems*, pages 482–489, 2009.

[35] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2 edition, 2006.

[36] B. McCloskey, F. Zhou, D. Gay, and E. A. Brewer. Autolocker: Synchronization inference for atomic sections. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–358, 2006.

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

[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[39] J. Sifakis. Deadlocks and livelocks in transition systems. In *Symposium on Mathematical Foundations of Computer Science*, pages 587–600, 1980.

[40] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345, 2006.

[41] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2010.

[42] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *LNCS*, pages 139–154, 2009.

[43] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–263, 2009.

A The Corrected Swapping Program in UPC

```
#include "upc.h"
#include <stdio.h>
shared int A[THREADS];
upc_lock_t *shared lk[THREADS];
shared int s[THREADS];
shared int setS[THREADS];
int i, temp;
int main()
{ int k;
if(MYTHREAD==0)
for(k=0;k<3;k++)
{ lk[k]=upc_global_lock_alloc();
 s[k]=0;
 setS[k]=0;}
upc_barrier(1);
// The body of each thread starts
A[MYTHREAD] = MYTHREAD;
i = MYTHREAD; /* i is a thread index.*/
if(MYTHREAD==0)  return 0;
if(MYTHREAD==1)
{
int cnt0;
int SS2;
for (cnt0=0;cnt0<10;cnt0++) {
 upc_memget(&SS2, &setS[2], sizeof (int));
 while (SS2 &&  (s[1] == 2)) upc_memget(&SS2, &setS[2], sizeof (int));
 s[1] = 2;
 setS[1] = 1;
 upc_lock(lk[1]);    /* Acquire locks */
 upc_lock(lk[s[1]]);
 upc_memget(&SS2, &setS[2], sizeof (int));
 while (!SS2) upc_memget(&SS2, &setS[2], sizeof (int));
 upc_unlock(lk[1]); /* Unlock */
 upc_unlock(lk[s[1]]);
 setS[1] = 0;
}
}
else if (MYTHREAD==2)
{
int cnt1;
int SS1;
int s1;
for (cnt1=0;cnt1<10;cnt1++) {
 upc_memget(&SS1, &setS[1], sizeof (int));
 while(!SS1) upc_memget(&SS1, &setS[1], sizeof (int));  /* It gives priority to Thread 1*/
 s[2] = 1;
 setS[2] = 1;
 upc_memget(&SS1, &setS[1], sizeof (int));
 upc_memget(&s1, &s[1], sizeof (int));
 while ((s1 != 2) || SS1) {
  upc_memget(&SS1, &setS[1], sizeof (int));
  upc_memget(&s1, &s[1], sizeof (int));}
 upc_lock(lk[2]);       /* Acquire locks */
 upc_lock(lk[s[2]]);
 /* Swap  */
 upc_unlock(lk[s[2]]); /* Unlock */
 upc_unlock(lk[2]);
 setS[2] = 0;
}
}
return 0;
}
```

Listing 4: The corrected swapping example described in Section 5.2

B Single Lane Bridge

```
byte procStamp=0;
byte occupied=0;
byte capacity=2;
byte nblue=0;
byte nred=0;
byte pc[2]=0;


active proctype RedCar(){
byte choice=0;
/* This part corresponds to while(true)
    in RedCar Class. */
do
::pc[0]==0->atomic{
 if
 ::(1)-> pc[0]= 1;
  ::else-> skip;
 fi;}

::pc[0]==1->atomic {
 /* Addition of one new red car to the bridge */

 if
 ::(capacity>occupied)->
  nred=nred+1; occupied=occupied+1;
 ::else->skip;
 fi;
  pc[0]= 2;
}
/*The folloing statement corresponds to RedExit()*/
::(pc[0]==2)->atomic
{
 if
 ::(nred>0)-> pc[0]= 3;
  ::else-> pc[0]= 4;
 fi
}
::(pc[0]==3)->atomic
{
 pc[0]= 4;
 /*Random decision for removal of a car*/
 if
 ::(1)->choice=1;
 ::(1)->choice=0;
 fi;
}
::(pc[0]==4)->atomic
{
 if
 /*Actual removal of a red car*/
 ::(choice==1|| capacity==nred)->
  nred=nred-1; occupied=occupied-1;choice=0;
 ::else->skip;
 fi;
  pc[0]= 5;
}
::(pc[0]==5)->{pc[0]=0;}
od
}
active proctype BlueCar(){
byte choice=0;
do
::pc[1]==0->atomic{
 if
 /* This part corresponds to while(true)
```

```
    in BlueCar Class. */
 ::(1)-> pc[1]= 1;
  ::else-> pc[1]= -1;
 fi;}

::pc[1]==1->atomic {
 if
 /* Addition of a blue car to the bridge  */
 ::(capacity>occupied)->
  nblue=nblue+1; occupied=occupied+1;
 ::else->skip;
 fi;
        pc[1]= 2;
}
/*The folloing statement corresponds to BlueExit()*/
::(pc[1]==2)->atomic
{
 if
 ::(nblue>0)-> pc[1]= 3;
  ::else-> pc[1]= 4;
 fi
}
::(pc[1]==3)->atomic
{
 pc[1]= 4;
 /*The following if statements simulates the random function*/
 if
 ::(1)->choice=1;
 ::(1)->choice=0;
 fi;
}

::(pc[1]==4)->atomic
{
 /*Actual removal of a blue car from the bridge*/
 if
 ::(choice==1|| capacity==nblue)->
  nblue=nblue-1; occupied=occupied-1;choice=0;
 ::else->skip;
 fi;
  pc[1]= 5;
}
::(pc[1]==5)->{pc[1]=0;}
od
}
```

Listing 5: Synchronization Skeleton of program in Figure 6.

```
class CorrectBridge extends Bridge {
...
    synchronized void redEnter() throws InterruptedException {
        if(onbridge==0)
  {++nred;
  onbridge++;
  if(procstamp==1)
   procstamp=3;
  else
   procstamp=2;
  }
    }

    synchronized void redExit(){
 --nred;
 --onbridge;
 notifyAll();
   }
    synchronized void redWait(){
```

```
 while(procstamp!=1)
  wait();}
    synchronized void blueEnter() throws InterruptedException {
         if(onbridge==0)
          {++nblue;
         onbridge++;
      if(procstamp==2)
  procstamp=3;
       else
  procstamp=1;

  }
    }
    synchronized void blueExit(){
        --nblue;
--onbridge;
 notifyAll();
    }
    synchronized void blueWait(){
 while(procstamp!=2)
  wait();}

}
```

Listing 6: Corrected Bridge in Java.

```
class NewBlueCar implements Runnable {
Bridge control;
...
    public void run() {
      try {
        while (true) {
            control.blueEnter();
            control.blueExit();
     control.blueWait();
         }
      } catch (InterruptedException e){}
    }
}
```

Listing 7: Corrected BlueCar.

```
class NewRedCar implements Runnable {
Bridge control;
....
public void run() {
 try {
   while(true) {
     control.redEnter();
     control.redExit();
     control.redWait();
       }
    } catch (InterruptedException e){}
   }
}
```

Listing 8: Corrected RedCar.

C Barrier Synchronization

```
mtype = { ready, exec, success };
mtype pc1=ready;
mtype pc2= ready;
mtype pc3= ready;

active proctype process1 ()
{
do
 pc1 = exec;
 pc1 = success;
 pc1 = ready;
od

}

active proctype process2 ()
{
do
 pc2 = exec;
 pc2 = success;
 pc2 = ready;

od

}

active proctype process3()
{
do
 pc3 = exec;
 pc3 = success;
 pc3 = ready;

od
}
```

Listing 9: Barrier Synchronization in Promela.

```
mtype = { ready, exec, success };
mtype pc1=ready;
mtype pc2= ready;
mtype pc3= ready;
active proctype process1 ()
{
do
::atomic{
 (pc1 == ready) &&
   (
   (pc2 == pc3) ||
   (pc2==exec) || (pc3==exec)
   )
  -> pc1 = exec;
 }
::atomic{
 (pc1 == exec) &&
   (
   (pc2 == pc3) ||
   (pc2==success) ||(pc3==success)
   )
   -> pc1 = success;
 }
::atomic{
 (pc1 == success)  &&
   (
```

```
   (pc2 == pc3) ||
   (pc2==ready) || (pc3==ready)
   )
  -> pc1 = ready;
 }
od
}

active proctype process2 ()
{
do
::atomic{
 (pc2 == ready) &&
  (
  (pc1 == pc3) ||
  (pc1==exec) || (pc3==exec)
  )
  -> pc2 = exec;
 }
::atomic{
  (pc2 == exec)
  (
  (pc1 == pc3) ||
  (pc1==success) ||(pc3==success)
  )
  -> pc2 = success;
 }
::atomic{
  (pc2 == success)  &&
  (
  (pc1 == pc3) ||
  (pc1==ready) || (pc3==ready)
  )

  -> pc2 = ready;
 }
od
}

active proctype process3()
{
do
::atomic{
  (pc3 == ready) &&
  (
  (pc1 == pc2) ||
  (pc1==exec) || (pc2==exec)
  )

  -> pc3 = exec;
 }
::atomic{
  (pc3 == exec) &&
  (
  (pc1 == pc2) ||
  (pc1==success) ||(pc2==success)
  )

  -> pc3 = success;
 }
::atomic{
  (pc3 == success)  &&
  (
  (pc1 == pc2) ||
  (pc1==ready) || (pc2==ready)
  )
  -> pc3 = ready;
```

```
 }
od
}
```

Listing 10: Barrier Synchronization in Promela after correction for ensuring safety.

```
mtype = { ready, exec, success };
mtype pc1=ready;
mtype pc2= ready;
mtype pc3= ready;
#define inv     ( ((pc1 != ready) && (pc2 != ready) && (pc3 != ready)) || \
  ((pc1 != exec) && (pc2 != exec) && (pc3 != exec)) ||\
  ((pc1 != success) && (pc2 != success) && (pc3 != success)) )
active proctype process1 ()
{
do
::atomic{
 inv && (pc1 == ready) &&
   (
    ((pc2 != success) && (pc3!=success)) ||
    ((pc2 == success) && (pc3==success))
   )

  -> pc1 = exec;
 }
::atomic{
 inv && (pc1 == exec) && (pc2 != ready) && (pc3 !=ready)
   -> pc1 = success;
 }
::atomic{
 inv && (pc1 == success)  &&
    ( ((pc2 == ready) && (pc3 ==ready)) ||
     ((pc2 == success) && (pc3 ==success)) )
  -> pc1 = ready;
 }
od
}

active proctype process2 ()
{
do
::atomic{
 inv && (pc2 == ready) &&
   (
    ((pc1 != success) && (pc3!=success)) ||
    ((pc1 == success) && (pc3==success))
   )
  -> pc2 = exec;
 }
::atomic{
 inv && (pc2 == exec) && (pc1 != ready) && (pc3 !=ready)

  -> pc2 = success;
 }
::atomic{
 inv && (pc2 == success)  &&
    ( ((pc1 == ready) && (pc3 ==ready)) ||
    ((pc1 == success) && (pc3 ==success)) )
  -> pc2 = ready;
 }
od
}

active proctype process3()
{
do
::atomic{
```

```
 inv && (pc3 == ready) &&
   (
    ((pc2 != success) && (pc1!=success)) ||
    ((pc2 == success) && (pc1==success))
   )

  -> pc3 = exec;
 }
::atomic{
 inv && (pc3 == exec) && (pc2 != ready) && (pc1 !=ready)
  -> pc3 = success;
 }
::atomic{
 inv && (pc3 == success)  &&
   ( ((pc2 == ready) && (pc1 ==ready)) ||
     ((pc2 == success) && (pc1 ==success)) )
  -> pc3 = ready;
 }
od
}
```

Listing 11: Barrier Synchronization in Promela after correction for ensuring of $allR \rightsquigarrow allS$.

D Token Ring

```
int x1 =1;
int x2 =2;
int x3 = 1;
active proctype process1 ()
{
do
  x1 = (x3+1) % 4;
od

}

active proctype process2 ()
{
do
 x2 = x1;
od
}


active proctype process3 ()
{
do
 x3 = x2;
od
}
```

Listing 12: Token Ring in Promela.

```
int x1 =1;
int x2 =2;
int x3 = 1;
active proctype process1 ()
{
do
::atomic{
  (x3 != ((x1+3) % 4)) -> x1 = (x3+1) % 4;
 }
od

}
active proctype process2 ()
{
do
::atomic{
  (x1 != x2)  -> x2 = x1;
 }
od
}
active proctype process3 ()
{
do
::atomic{
 (x3 != x2)  -> x3 = x2;
 }
od
}
```

Listing 13: Token Ring after correction to ensure $true \rightsquigarrow (x1 = x2 \wedge x2 = x3)$.

E Two Single Lane Bridges

```
byte choice[2]=0;
int br[2]=0;
byte total[2]=0;
byte nblue[2]=0;
byte nred[2]=0;
byte occupied[2]=0;
byte capacity=0;
byte pc[2]=0;

active [2] proctype TwoBridges() {
do
::(pc[_pid]==0)->  pc[_pid]= 1;
::(pc[_pid]==1)->atomic
{
/*Each active process simulates either of threads (RedCar or BlueCar)
  if _pid=0 it simulates RedCar oterwise it simulates Bluecar*/
 if
 ::(_pid==0)-> pc[_pid]= 2;
 ::else-> pc[_pid]= 6;
 fi
}

::(pc[_pid]==2)->atomic{
 /*random selection of bridge*/
 if
   ::(1)-> br[_pid]=0;
   ::(1)-> br[_pid]=1;
 fi;
 pc[_pid]= 3; }

::(pc[_pid]==3)->atomic{
          if
 /*A new red car is inserted into bridge br[_pid]*/
 ::(capacity>occupied[br[_pid]])->{
  nred[br[_pid]]=nred[br[_pid]]+1;
  occupied[br[_pid]]=occupied[br[_pid]]+1;]
  total[_pid]=total[_pid]+1;}
 ::else->skip;
 fi;
  pc[_pid]= 4;
}

::(pc[_pid]==4)->atomic{
  pc[_pid]= 5;
  /*Random choice for removal form the bridge */
  if
  ::(1)-> choice[_pid]=1;
  ::(1)->choice[_pid]=0;
  fi;
 }
::(pc[_pid]==5)->atomic
{     if
 /*Actual removal form the bridge */
 ::((choice[_pid]==1&&nred[br[_pid]]>0)|| capacity==nred[br[_pid]])->
  {nred[br[_pid]]=nred[br[_pid]]-1;
   occupied[br[_pid]]=occupied[br[_pid]]-1;
   choice[_pid]=0; total[_pid]=total[_pid]-1;}
 ::else->skip;
 fi;

 pc[_pid]= 1;
}

::(pc[_pid]==6)->atomic{ pc[_pid]= 7;}
```

```
::(pc[_pid]==7)->atomic{
 pc[_pid]= 8;
 /*random selection of bridge*/
 if
 ::(1)-> br[_pid]=0;
 ::(1)-> br[_pid]=1;
 fi;}

::(pc[_pid]==8)->atomic{
 /*A new blue car is inserted into bridge br[_pid]*/
 if
          ::(capacity>occupied[br[_pid]])->
   {nblue[br[_pid]]=nblue[br[_pid]]+1;
   occupied[br[_pid]]=occupied[br[_pid]]+1;
   total[_pid]=total[_pid]+1;}
 ::else->skip;
 fi;
 pc[_pid]=9;
 }

::(pc[_pid]==9)->atomic{
 pc[_pid]= 10;
 /*Random choice for removal form the bridge */
 if
 ::(1)-> choice[_pid]=1;
 ::(1)-> choice[_pid]=0;
 fi;
 }
::(pc[_pid]==10)->atomic{
 /*Actual removal form the bridge */
          if
 ::((choice[_pid]==1&&nblue[br[_pid]]>0)|| capacity==nblue[br[_pid]])->
   {nblue[br[_pid]]=nblue[br[_pid]]-1;
    occupied[br[_pid]]=occupied[br[_pid]]-1;
    choice[_pid]=0; total[_pid]=total[_pid]-1;}
 ::else->skip;
 fi;
  pc[_pid]= 7;
 }

od
}
```

Listing 14: Two Single Lane Bridges in Promela.

```
byte choice[2]=0;
int br[2]=0;
byte total[2]=0;
byte nblue[2]=0;
byte nred[2]=0;
byte occupied[2]=0;
byte capacity=0;
byte pc[2]=0;
active [2] proctype TwoBridges() {

do
::(pc[_pid]==0)->  pc[_pid]= 1;
::(pc[_pid]==1)->atomic
{

 if
 ::(_pid==0)-> pc[_pid]= 2;
 ::else-> pc[_pid]= 6;
 fi
}

::(pc[_pid]==2)->atomic{
```

```
  if
  ::(1)-> br[_pid]=0;
  ::(1)-> br[_pid]=1;
  fi;
 pc[_pid]= 3; }

::(pc[_pid]==3)->atomic{
          if
 ::(capacity>occupied[br[_pid]]&&nblue[br[_pid]]==0)->{
    nred[br[_pid]]=nred[br[_pid]]+1;
    occupied[br[_pid]]=occupied[br[_pid]]+1; total[_pid]=total[_pid]+1;}
 ::else->skip;
 fi;
  pc[_pid]= 4;
}

::(pc[_pid]==4)->atomic{
  pc[_pid]= 5;
  if
  ::(1)-> choice[_pid]=1;
  ::(1)->choice[_pid]=0;
  fi;
 }
::(pc[_pid]==5)->atomic
{     if
 ::((choice[_pid]==1&&nred[br[_pid]]>0)|| capacity==nred[br[_pid]])->
  {
  nred[br[_pid]]=nred[br[_pid]]-1; occupied[br[_pid]]=occupied[br[_pid]]-1;
  choice[_pid]=0; total[_pid]=total[_pid]-1;
  }
 ::else->skip;
 fi;

 pc[_pid]= 1;
}

::(pc[_pid]==6)->atomic{ pc[_pid]= 7;}

::(pc[_pid]==7)->atomic{
 pc[_pid]= 8;
 if
 ::(1)-> br[_pid]=0;
 ::(1)-> br[_pid]=1;
 fi;}

::(pc[_pid]==8)->atomic{
 if
          ::(capacity>occupied[br[_pid]]&&nred[br[_pid]]==0)->
    { nblue[br[_pid]]=nblue[br[_pid]]+1;
    occupied[br[_pid]]=occupied[br[_pid]]+1; total[_pid]=total[_pid]+1;
   }
 ::else->skip;
 fi;
 pc[_pid]=9;
 }

::(pc[_pid]==9)->atomic{
 pc[_pid]= 10;
   if
   ::(1)-> choice[_pid]=1;
   ::(1)-> choice[_pid]=0;
   fi;
 }
::(pc[_pid]==10)->atomic{
          if
 ::((choice[_pid]==1&&nblue[br[_pid]]>0)|| capacity==nblue[br[_pid]])->
   {
```

```
    nblue[br[_pid]]=nblue[br[_pid]]-1;
    occupied[br[_pid]]=occupied[br[_pid]]-1; choice[_pid]=0; total[_pid]=total[_pid]-1;
    }
 ::else->skip;
 fi;
  pc[_pid]= 7;
 }

od
}
```

Listing 15: Two Single Lane Bridges in Promela after correction to ensure safety.

```
byte choice[2]=0;
int br[2]=0;
byte total[2]=0;
byte nblue[2]=0;
byte nred[2]=0;
byte occupied[2]=0;
byte capacity=0;
byte pc[2]=0;
active [2] proctype TwoBridges() {

do
::(pc[_pid]==0)->  pc[_pid]= 1;
::(pc[_pid]==1)->atomic
{

 if
 ::(_pid==0)-> pc[_pid]= 2;
 ::else-> pc[_pid]= 6;
 fi
}

::(pc[_pid]==2)->atomic{
  if
  ::(1)-> br[_pid]=0;
  fi;
 pc[_pid]= 3; }

::(pc[_pid]==3)->atomic{
          if
 ::(capacity>occupied[br[_pid]]&&nblue[br[_pid]]==0)->{
  nred[br[_pid]]=nred[br[_pid]]+1;
  occupied[br[_pid]]=occupied[br[_pid]]+1;
  total[_pid]=total[_pid]+1;}
 ::else->skip;
 fi;
  pc[_pid]= 4;
}

::(pc[_pid]==4)->atomic{
  pc[_pid]= 5;
  if
  ::(1)-> choice[_pid]=1;
  ::(1)->choice[_pid]=0;
  fi;
 }
::(pc[_pid]==5)->atomic
{     if
 ::((choice[_pid]==1&&nred[br[_pid]]>0)|| capacity==nred[br[_pid]])->
          {nred[br[_pid]]=nred[br[_pid]]-1;
      occupied[br[_pid]]=occupied[br[_pid]]-1;
      choice[_pid]=0; total[_pid]=total[_pid]-1;}
 ::else->skip;
 fi;
```

```
 pc[_pid]= 1;
}

::(pc[_pid]==6)->atomic{ pc[_pid]= 7;}

::(pc[_pid]==7)->atomic{
 pc[_pid]= 8;
 if

 ::(1)-> br[_pid]=1;
 fi;}

::(pc[_pid]==8)->atomic{
 if
          ::(capacity>occupied[br[_pid]]&&nred[br[_pid]]==0)->
  {nblue[br[_pid]]=nblue[br[_pid]]+1;
  occupied[br[_pid]]=occupied[br[_pid]]+1;
  total[_pid]=total[_pid]+1;}
 ::else->skip;
 fi;
 pc[_pid]=9;
 }

::(pc[_pid]==9)->atomic{
 pc[_pid]= 10;
 if
 ::(1)-> choice[_pid]=1;
 ::(1)-> choice[_pid]=0;
 fi;
 }
::(pc[_pid]==10)->atomic{
           if
 ::((choice[_pid]==1&&nblue[br[_pid]]>0)|| capacity==nblue[br[_pid]])->
  {nblue[br[_pid]]=nblue[br[_pid]]-1;
   occupied[br[_pid]]=occupied[br[_pid]]-1;
   choice[_pid]=0; total[_pid]=total[_pid]-1;}
 ::else->skip;
 fi;
  pc[_pid]= 7;
 }

od
}
```

Listing 16: Two Single Lane Bridges in Promela after correction to ensure progress property.