Towards Algorithmic Correction of Concurrency Failures

Mohammad Amin Alipour
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
malipour@mtu.edu

Ali Ebnenasir
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

ABSTRACT
Designing inter-thread synchronization mechanisms is an important part of concurrent programming. Poor design of synchronization may result in concurrency failures such as data race, deadlock or starvation. In this paper, we present our ongoing work on algorithmic revision of inter-thread synchronization mechanisms, known as synchronization skeletons. Our preliminary results illustrate that such algorithmic revisions facilitate the correction of concurrency failures. We also present a method for reducing the algorithmic correction of starvation to correction of safety violations. We illustrate the proposed approach in the context of the Dining Philosophers problem.

Keywords
Concurrency Failures, Inter-Thread Synchronization, Program Synthesis

1. INTRODUCTION
Design and debugging of concurrent programs are complex tasks, in part due to inherent non-determinism and the crosscutting nature of mutual exclusion and progress (deadlock/starvation-freedom) requirements. As such, manually fixing concurrency failures is an error-prone task which often results in new design flaws. In this paper, we present our vision for algorithmic correction of synchronization skeletons towards correcting concurrency failures, where a synchronization skeleton [5] represents part of the program functionalities related to inter-thread synchronizations. We also present a method for reducing starvation failures to safety failures, thereby enabling the reuse of our previous work [1, 11] on algorithmic correction of safety failures for the correction of starvation.

There are numerous techniques for detection and correction of concurrency failures. For example, some researchers propose techniques for detection of data races [13] and tolerating them [18, 17]. Flanagan and Freund [14] use type systems to enable atomicity in concurrent programs. Specification-based synthesis approaches [6] generate synchronization from formal specifications. In these methods, if a new requirement (e.g., deadlock-freedom) is introduced, then synthesis should be restarted from a new specification. As such, it is difficult to reuse an existing program towards correcting it. Wang et al. [20] utilize a control-theoretic approach to resolve deadlocks. Jobstmann et al. [15] present a game-theoretic approach for fixing faults. Vechev et al. [19] infer synchronization mechanisms from safety requirements. These techniques are essentially specification-based methods. Katz and Peled [16] propose a genetic programming approach for correction of programs, which is based on search and verification rather than revision of an existing program.

We propose an alternative paradigm for the correction of concurrency failures based on the concept of satisfying subsets. For a program \( p \) that meets part of its specification, denoted \( \text{spec}_p \), but fails to satisfy a specific data race-freedom or progress requirement, a Satisfying Subset (SS) [10] is a subset of the inter-thread synchronization traces of \( p \) that meets the data race-freedom or progress requirement while preserving \( \text{spec}_p \). In the proposed approach, we utilize program slicing techniques to extract the synchronization skeleton of programs. Afterwards, we generate a behavioral model either by explicit state enumeration or using Binary Decision Diagrams (BDD) [3]. Subsequently, we search for SSs. If an SS exist, then it represents a corrected version of the synchronization skeleton. Otherwise, we declare that the concurrency failure cannot be corrected by revising synchronization skeletons. Our previous work illustrates that correcting progress failures of multiple threads is a hard problem [10, 11]. In this paper, we present a technique that reduces the starvation-freedom of multiple threads to ensuring safety, for which we already have designed efficient algorithms [10, 1, 11]. We use the Dining Philosophers (DP) [7] problem to demonstrate our approach.

Organization. Section 2 represents the basic concepts of our ongoing work on SSs. Section 3 presents a method for the reduction of starvation-freedom to safety (in the context of the DP problem [7]). Section 4 discusses open problems and future directions.

2. PRELIMINARIES
In this section, we represent the problem of algorithmic revision of synchronization skeletons (i.e., finding SSs) adapted from [10]. Moreover, Sections 2.1 and 2.2, respectively discuss the correction of safety and progress failures.

Synchronization Skeleton. Consider a multi-threaded...
program $p$ with a fixed number of threads. We represent a synchroniz-ation skeleton of $p$ as a finite state machine $M_p = \langle S_p, \delta_p, I_p \rangle$, where $S_p$ denotes a finite set of synchroniza-tion states, where a synchronizat-ion states are a unique value of synchronization variables (e.g., lock variables, semaphores) of $p$. $\delta_p$ represents a finite set of transitions in $S_p \times S_p$. A transition from state $s_0$ to $s_1$, denoted $(s_0, s_1)$, is enabled at $s_0$ iff $(s_0, s_1) \in \delta_p$. $I_p$ is a finite set of initial states, which represents the initial values of synchroniz-a-tion variables.

A state predicate is a subset of $S_p$. A state predicate $X$ is true (i.e., holds) in a state $s \in X$. A computation (i.e., synchronization trace) is a sequence of states that is either infinite, or if it is finite, then no transition of $M_p$ is enabled in its final state, called a halting state. A computation prefix from state $s_0$ to state predicate $X$ with length $k$ is a finite sequence $\sigma = s_0, s_1, \ldots, s_k$ of states, where each transition $(s_i, s_{i+1})$ is in $\delta_p$, $0 \leq i < k$ and $s_k \in X$. A state $s_k$ is a deadlock state iff no transition of $M_p$ is enabled at $s_k$. Since algorithmic correction we may remove transitions and create new deadlock states, we distinguish such states from halting states by stuttering at the halting states. Hereafter, we use the terms “state and “synchronization state”, and “program” and “synchronization skeleton” interchangeably.

Safety Property. Intuitively, a safety property stipulates that nothing bad ever happens. Formally, we represent a safety property by a Linear Temporal Logic (LTL) [12] formula $P$, where $P$ represents a set of good states (e.g., set of deadlock-free or data race-free states) and denotes the always temporal operator. A computation $\sigma = s_0, s_1, \ldots$ satisfies a safety property $P$ from an initial state $s_0$ iff for any state $s_i$ in $\sigma$, where $i \geq 0$, we have $s_i \in P$.

Progress Property. Progress requirements capture the notion that something good happens eventually. For exam-ple, each thread that tries to access shared variables (i.e., enter a critical section of its code) will eventually be able to do so. We use “leads-to” (denoted $\leadsto$) [4] properties to specify the progress requirements of each thread. A computation $\sigma = s_0, s_1, \ldots$ satisfies a progress property $P \leadsto Q$ ($P$ and $Q$ are state predicates) from $s_0$ iff for any state $s_i$ in $\sigma$, where $i \geq 0$, if $s_i \in P$, then there exist a state $s_j$, where $j \geq i$, such that $s_j \in Q$. A program $M_p$ satisfies its specification $spec$ from $I_p$ iff all computations of $M_p$ satisfy safety and progress properties of $spec$ from $I_p$.

Correction Problem. Consider a program $M_p = \langle S_p, \delta_p, I_p \rangle$ that satisfies a specification $spec_p$ from $I_p$, but does not satisfy a safety/progress property $P$ from $I_p$. Identify a revised version of $M_p$, denoted $M_{p'} = \langle S_p, \delta_p', I_{p'} \rangle$, such that (1) $S_p = S_p$; (2) $I_p = I_{p'}$; (3) $\delta_p' \subseteq \delta_p$, and (4) $M_{p'}$ satisfies $spec_{p'} \land P$ from $I_p$. (The synthesized program $M_{p'}$ captures a SS.)

2.1 Correction of Safety Failures

Our algorithm in [11] for the correction of safety failures (such as data races and deadlocks) includes a fixpoint com-putation that eliminates any reachable bad state $s_b$ where $\neg P$ holds, for a safety property $P$. We eliminate $s_b$ by removing any transition reaching $s_b$; i.e., making $s_b$ unreachable. As such, we may introduce new deadlock states, which are removed recursively. Such state elimination continues until no more bad states are reachable, or an initial state becomes deadlocked, where we declare failure; i.e., it is impossible to ensure $P$ while preserving $spec_p$.

2.2 Correction of Progress Failures

The failure of a program $M_p$ to satisfy a property $P \leadsto Q$ has two reasons, namely reachable deadlocks and reachable non-progress cycles. For example, if $M_p$ has a progress fail-ure for a thread $T$ such that $T$ tries to enter its critical section, but it may never be given the chance to do so, then the cause of this failure is two-fold: either $T$ deadlocks or some other non-deadlocked threads deprive $T$ from entering its critical section by a cyclic execution schedule among themselves, called a non-progress cycle. That is, $T$ is starved. In our previous work [11, 1], we present an al-gorithm for correcting failures of satisfying $P \leadsto Q$. First, our algorithm removes deadlock states using the algorithm given in Section 2.1. If no initial state is removed, the re-sulting synchronization skeleton is fed to another algorithm for correcting non-progress cycles as follows. We first determine a rank for any reachable state, where rank of a state $s$, denoted $\text{Rank}(s)$, is the length of the shortest computa-tion prefix from $s$ to $Q$. Afterwards, any state in $P$ that has a rank equal to $\infty$ is made unreachable, where the rank $\infty$ means that there is no computation prefix from $s$ to $Q$. Subsequently, we eliminate any transition $(s_0, s_1)$, where $(s_0 \notin Q) \land (\text{Rank}(s_0) \leq \text{Rank}(s_1))$.

We have demonstrated [11, 1] the soundness and completeness of this algorithm for a single property $P \leadsto Q$ (in the state space of the input program). However, there are a few caveats. First, if one has to correct the progress failures of multiple threads, then our algorithm should be applied for each leads-to property one at a time in a stepwise fashion. Second, our algorithm may remove transitions that are useful for correcting the non-progress failures of another leads-to property. As such, in a stepwise method, the order of correction may affect the success of correction. Thus, it is desirable to devise heuristics that enable the correction of multiple progress failures while removing minimal num-ber of transitions. Next section presents a heuristic for such corrections.

3. REDUCTION OF STARVATION TO SAFETY VIOLATION

In this section, we present a heuristic for reducing the problem of correcting the starvation failures of multiple pro cesses to the problem of ensuring safety. Given a program with multiple starvation failures, we first use computation superimposition [4] to build an intermediate program which manifests starvation failures as a safety violation. Then, we automatically correct the intermediate program to ensure the safety property, thereby guaranteeing starvation-freedom. We demonstrate our approach in the context of the classic example of Dining Philosophers (DP) [9].

3.1 Example: Dining Philosophers Problem

The DP program [9] includes $n$ processes representing $n$ philosophers sitting at a round table and continuously chang-ing their state between thinking, hungry and eating. There are just $n$ chopsticks on the table placed in between adjacent philosophers. A philosopher needs both left and right chopsticks to be able to eat. Figure 1 illustrates a pseudo code of the philosopher $i$. The variable $c$ denotes the state of philosopher $i$ with a domain of three values $\{0, 1, 2\}$ respectively representing the states of thinking, hungry and eating. The global array chopsticks provides bi-
Figure 1: Code of the i-th philosopher with n philosophers at the round table.

32. Intermediate DP Program

Figure 2 presents the superimposed program. This program utilizes two new global variables CS and starvation (Lines 2 and 3 in Figure 2), where CS denotes the number of processes entered their critical sections (philosophers that ate), in the current iteration, and starvation is a flag for situations that starvation may arise. Each philosopher increments CS atomically when (s)he eats (see the atomic block in Line 16), thereby preventing data races on CS. When process i finishes an iteration and wants to start a new iteration, it checks (Lines 20) if there are other processes that have not eaten yet. In this case, the continuation of process i may lead to starvation of other processes, thus, Line 21 sets the global variable starvation to indicate the starvation failure. Line 22 decrements the value of CS for the next iteration. Therefore, if we ensure that the safety property \( \text{starvation} = \text{false} \) holds, then that implies that all processes satisfy their progress requirements.

3.3 Correction of Progress Failure for DP

In this section, we correct the superimposed program in Figure 2 to ensure the safety requirement \( \text{starvation} = \text{false} \). We apply our correction algorithm (described in Section 2.1) in a case with five philosophers \( n = 5 \) to automatically find a satisfying subset for the safety requirement. Towards this end, we used a software tool we have developed for the correction of concurrency failures, called the Correction Synthesizer (CorSyn). The corrected program meets the following constraints: (1) At least one process chooses a different order in acquiring chopsticks. This condition breaks the resource demand cycle that leads to a deadlock, and (2) At the end of each iteration, each process must ensure that other processes enter their critical sections. This condition prohibits starvation of processes. (To gain more confidence in the implementation of CorSyn, we have model checked the corrected program with the SPIN model checker. The Promela model is available at http://cs.mtu.edu/~malipour/SynchCor/dpp.pml.)

Figure 3 illustrates the corrected program. For imposing the first constraint, the conditional statement in Lines 8-11 has been automatically synthesized, which forces the philosopher at position 0 to choose a different order for acquiring chopsticks from others. Line 19 ensures the second constraint, by delaying a process until others enter their critical sections. This statement can be realized by barrier synchronization constructs in programming languages.
4. OPEN PROBLEMS AND FUTURE WORKS

The essence of the notion of satisfying subsets is based on identifying the non-determinism that causes concurrency failures, which we call the unnecessary non-determinism. The necessary non-determinism plays a major role in the success of our approach. If the faulty program does not contain sufficient non-determinism, then pruning the unnecessary transitions could either fail quite often, or result in sequential ordering of processes, which are not desirable. For example, in the DP program presented in Section 3.1, if philosophers use the same ordering for grabbing chopsticks, this approach will result in a highly sequential ordering for ensuring the progress of each philosopher. We are currently investigating techniques that add necessary non-determinism to programs automatically. For example in the DP program, the random order of acquiring chopsticks (see Lines 8-9 of Figure 1) does not affect mutual exclusion requirement, and enables our algorithms to derive a solution.

We also plan to study the mutual impact of the read restrictions imposed on each process (with respect to the local state of other processes) and the degree of parallelism in concurrent programs. Specifically, previous work [2] demonstrates that if processes have read restrictions (i.e., limited observability) with respect to the local state of other processes, then the complexity of correction is exponential in program state space. Nonetheless, a higher degree of parallelism and the limits of observability. We therefore is desirable to strike a balance between the degree of parallelism and the limits of observability. We are also working on improving the scalability of our correction algorithms by developing customized data structures for correction and by parallelizing the correction algorithms.

5. REFERENCES


