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
Concurrent Failures, Inter-Thread Synchronization, Program Synthesis

1. INTRODUCTION

Designing 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 [19, 18]. 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. [21] utilize a control-theoretic approach to resolve deadlocks. Jobstmann et al. [15] present a game-theoretic approach for fixing faults. Vechev et al. [20] 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 specifications, denoted $spec_p$, 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 $spec_p$. 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, 11, 12]. We use the Dining Philosophers (DP) [7] problem to demonstrate our approach.

Organization. Section 2 introduces our ongoing work using 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. SATISFYING SUBSETS: A CORRECTION APPROACH

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

Synchronization Skeleton. Consider a multi-threaded...
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 failure 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 thread deprives \( T \) 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 algorithm 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 resulting 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 computation 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 number 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 processes 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 changing 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 binary semaphores, as in \([8]\), for acquiring chopsticks. Local
1 int chopsticks[n] := 0;
2 process philosopher[i]
3 \{ int C := 0; /* thinking */
4 int first := 0, second := 0;
5 while (true)
6 \{ C := 1; /* hungry */
7 /* random ordering for grabbing chopsticks */
8 \{ if (1) then first := 1; second := (i+1)%n;
9 :: if (1) then first := (i+1)%n; second := 1;
10 \}
11 lock(chopsticks[first]);
12 lock(chopsticks[second]);
13 C := 2; /* eating */
14 unlock(chopsticks[first]);
15 unlock(chopsticks[second]);
16 C := 0; /* thinking */
17 \}

Figure 1: Code of the i-th philosopher with n philosophers at the round table.

Variables first and second denote the first chopstick and the second chopstick that philosopher \( i \) must acquire in order. In Lines 7-9, a hungry philosopher randomly selects an order of grabbing chopsticks. The possible orders include the acquirement of either the left chopstick first and then the right chopstick (left-right), or in reverse (right-left). In Lines 11 and 12, philosopher \( i \) tries to acquire the locks for the chopsticks in the specified order. After acquiring both chopsticks, philosopher \( i \) can eat (Line 13). Then, in Lines 14 and 15, \( i \) releases the chopsticks and re-enters the state of thinking (Line 16). This program prevents simultaneous accesses to shared resources by using lock() functions (i.e., safety is guaranteed). However, if the philosophers get hungry simultaneously and acquire the right chopsticks, all of them will starve. Hereafter, in this section, we use the terms “process” and “philosopher” interchangeably.

3.2 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 \( \square (\text{starvation} = \text{false}) \) 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 \( \square (\text{starvation} = \text{false}) \). 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 designed and implemented for the correction of concurrency failures, called the Correction Synthesizer (CorSyn). The corrected program meets the following constraints: (1) At least one
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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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 can be achieved if processes need not read too many global data structures and are mostly working locally. For example, in the DP program, any solution without any global data structures and are mostly working locally. 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.