Using Search to Synthesize a Checkpoint Program

Ian Dardik

Summer 2021

1 Intro

This note is the formal description of the checkpoint program synthesis problem and my plan for implementation. Here is a high level description of the program:

A user wants to create a distributed program and already has a good idea of what their desired algorithm looks like. However, the user does not want to write the algorithm by hand because they don't want to debug a distributed program. Instead, the user can input a precise statement of requirements to the synthesizer to produce a program that is guaranteed to satisfy all the given requirements. The more precise the given requirements are, the more likely we are to synthesize a program that the user desires.

2 Background

2.1 The nested exponential search in program synthesis

We can synthesize a program for p processes using the following BFS algorithm:

```
input: requirements
queue = [noop]
while true:
  program = pop(queue)
  for instruction in AllInstructions:  // branching exp
    newProgram = program + instruction
    if satisfiesAllRequirements(newProgram, requirements):  // interlreaving exp
      return newProgram
    if noSafetyViolations(newProgram, requirements):  // interlreaving exp
      add(queue, newProgram)
```

Note that there are two nested exponentials here:

1. The branching factor that is $O(b^n)$, where $b = |AllInstructions|$ (the number of available instructions) and n is the number of lines in the program
2. The time it takes to check all interleavings of the p processes to make sure that all requirements are satisfied in a given program.

I believe these are the two nested exponentials mentioned in [1], however I may be wrong. Either way, this is clearly very expensive.

2.2 Checkpointing

We will try to synthesize a distributed program, but we will restrict our search to only synthesize a checkpoint program. A checkpoint program is a distributed program that has checkpoipnts every few lines of code. A checkpoint is simply a barrier that synchronizes all processes. In this note we'll attempt to build checkpoint programs that have a checkpoint after *every* line of code whenever possible; we will only delay a checkpoint when we need to prevent a deadlock. Synthesizing a checkpoint program is easier than synthesizing a general distributed program because requirement-satisfaction checking only needs to happen between checkpoints. If we restrict our possible programs to have at most N lines of code between checkpoints and hold the number of processes p constant, then the number of interleavings we must check between any two checkpoints is bounded by some constant $il(N, p)$ (the number of interleavings of N lines of code for p processes). Thus checking requirement-satisfaction for a program of length n becomes $O(il(N, p) * n)$ instead of $O(il(n, p))$; that is, linear instead of exponential in terms of n.

Naturally, we incur several disadvantages with this method:

1. Restricting the number of lines without a checkpoint to N can cause the algorithm to miss potential solutions (programs that satisfy our requirements), however the gain in efficiency may prove to make this a worthwhile tradeoff.

2. Checkpoints reduce the parallelism of each process and can slow down the program we synthesize. My hope is that once we have produced a checkpoint program, it will be possible to perform some optimization by deleting unnecessary checkpoints to speed up the program.

3. Although we reduced the nested exponential to a linear check, the branching factor still yields an overall exponential runtime for our search algorithm. It may be interesting to perform some analysis to see how many branches we prune because of safety violations, and see if/when it affects the asymptotic runtime of the algorithm.

3 Formal Problem Description

3.1 The Problem

The inputs are:

1. State variables.

2. Initial state.

3. Safety requirements.

4. Liveness requirements.

5. The set of all possible instructions in the program.

6. "Decisions": some variables may have state transitions that are optional, and thus any instruction that takes the variable on this transition will be a "decision" by the user.

7. It is very likely I'll allow additional inputs in the future.

Given these inputs, the program will either output a program that satisfies the given requirements or will time out with no solution. For the near term future we will output a checkpoint program, however it would be nice to eventually optimize the synthesized program to remove as many checkpoints as possible.

3.2 States

State variables are described by a domain. We produce the domain of all possible states as the cartesian product of the domain across all state variables. As part of the program input the user must label each variable as global or local. Note that because the state variables are given, we can view our problem as a variant of a protocol completion problem.

3.3 Input Language

The safety and liveness requirements will be specified using an input language that will likely resemble TLA+. The input language is very important because it defines the expressive power that the user has to describe a specific program; the richer the language, the more likely a user can specify a program that they want and will find useful. Implementing each language feature may not be easy, especially since operators ought to be composable. For the time being I will support AND, OR, $\sim>$ (leads to), and expressions that test variable values (i.e. flag[p] = TRUE). In the future I may attempt to support additional features such as IF-THEN-ELSE, ALWAYS, EVENTUALLY, and fairness.

3.4 Instruction Set

The list of all possible instructions that the program can execute. This replaces the previous idea of a "transition constraint".

3.5 Decisions

A *decision* is tool for a user to specify a rule r as a decision $Dec(r)$. For example, if the rule $r = \forall p \in ProcSet : \neg flag[p] - - > flag[p]$, then $Dec(r)$ means that any instruction that changes a process' flag from FALSE to TRUE is considered a *decision instruction.* Any instruction that is executed after a decision instruction must check the current state for validity, but also the state identical to the current state except all other processes $q \neq p$ flags are set to FALSE. The idea here is that a process flipping its flag from FALSE to TRUE is a *decision* and does not have to happen at all for any given process.

This concept will support my needs for now, but I suspect it is not general enough to keep as an input in the long term. I will most likely need to extend decisions or replace them by a more general and powerful concept in the future.

3.6 Example

My first goal is to synthesize a program that's essentially equivalent to Peterson's Mutual Exclusion Algorithm so I will present this as an example. I believe that the following inputs will synthesize a program that's "close enough" to Peterson's. Assume $ProcSet = \{0, 1\}$, then:

1. State variables: cs (critical section): local, flag: global, turn: global. Their type signatures are:
   $\forall p \in ProcSet : cs[p] \in \{TRUE, FALSE\}$
   $\forall p \in ProcSet : flag[p] \in \{TRUE, FALSE\}$
   $turn \in ProcSet$

2. Initial state: $turn = 0$, $\forall p \in ProcSet : \neg cs[p] \wedge \neg flag[p]$

3. Safety requirements:

   (a) $\forall p, q \in ProcSet : cs[p] \wedge cs[q] => p = q$ (mutual exclusion)

   (b) $\forall p \in ProcSet : cs[p] => flag[p]$

4. Liveness requirements:
   (a) Required state: $\forall p \in ProcSet : cs[p]$. This just tells the program that at some point each process must enter the critical section.
   (b) $\forall p \in ProcSet : flag[p] \ \sim> cs[p]$
   (c) $\forall p \in ProcSet : flag[p] \ \sim> \neg flag[p]$
   (d) $\forall p \in ProcSet : cs[p] \ \sim> \neg cs[p]$
5. Transition constraint: one variable may change at a time
6. "Decisions": $\forall p \in ProcSet : Dec(\neg flag[p] - - > flag[p])$. This specifies that if the flag changes from FALSE to TRUE for a process, it is a *decision*.

Given these inputs, I believe that the program should be able to synthesize an algorithm that is fairly close to Peterson's algorithm. In particular, these inputs will avoid synthesizing a round-robin algorithm where each process *must* enter the critical section in each round because:

1. A process p sets its flag to TRUE, showing its intent to enter the critical section
2. p must enter the critical section, given by liveness requirements (a) and (b). Also note that $flag[p]$ when p enters the critical section by safety requirement (b).
3. p must be able to enter the critical section while $\forall q \in ProcSet : q \neq p, \neg flag[q]$ because setting a process' flag to TRUE is a *decision*.

As a result, we must synthesize an algorithm where one process p can enter the critical section while all other processes q do not intend to enter the critical section.

Unfortunately, the synthesized algorithm is only guaranteed to satisfy one of the the three required properties that are required to solve the critical section problem [2]:

1. Satisfies Mutual Exclusion: given by the safety property (a)
2. Does not satisfy Progress: the given requirements still allow us to synthesize a program that lets processes into the critical section in a round-robin fashion based on the *turn* global variable. This means that all processes can be a part of deciding who enters the critical section and when.
3. Does not satisfy Bounded Waiting: we will synthesize a program that *does* have a bounded amount of time for how long it takes a process to enter the critical section once it expresses intent; however, that time bound is not guaranteed to be a function of the number of processes.

I will need to think more about how to express the Progress and Bounded Waiting properties so we can synthesize programs with these guarantees. This may involve adding more features to the input languages, and potentially altering or replacing decisions. I plan on figuring out how to express the Progress propety before implementing the synthesizer; I do not plan on figuring out how to express Bounded Waiting before implementing the synthesizer (although it would be nice if I could figure this out).

4 Implementation

4.1 BFS

Implicitly abides by Occams Razor

4.2 Loops

Loop conditions will be branched by loop invariants, not by the conditions themselves.

References

[1] Rajeev Alur, Stavors Tripakis (2017). ACM SIGACT News, Volume 48, Issue 1. pp 55–90

[2] Wikipedia: Peterson's algorithm. `https://en.wikipedia.org/wiki/Peterson%27s_algorithm`