Using Search to Synthesize a Distributed Program

Ian Dardik

Summer 2021

1 Intro

This note is the formal description of the program synthesis problem I am trying to solve as well as a proposal for implementation. Here is a high level description of the problem:

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 do not 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. The user can think of developing a requirement spec for the synthesizer as analogous to writing a program; they may need to go through an iterative process of testing and updating their requirement spec to achieve their desired program. However, unlike writing a program in a programming language, users can guarantee their desired requirements will hold by construction.

2 Formal Problem Description

2.1 The Problem

The inputs are:

1. State variables.
2. Initial state.
3. Required states.
4. Safety requirements. Deadlocks are automatically considered a violation.
5. Liveness requirements.
6. The set of all possible instructions in the program.

Given these inputs, the program will either output a program that satisfies the given requirements, an error, or will time out with no solution. Search continues due to the following three reasons:

1. There are still states left on the queue. No states on the queue means that all possible programs have safety violations and we return an error.
2. There are still required states that haven't been visited.
3. There are still outstanding liveness requirements.

If we find a program that visits all required states and has no outstanding liveness requirements then we return success.

2.2 States

The user will provide the name of each state variable as well as its domain. Note that because the state variables are given, we can view our problem as a variant of a protocol completion problem. We may have the user to mark variables with permissions (e.g. rwrw where the 1st two bits are permissions for other processes and the 2nd two bits are permissions for a process' own variables), but this may not be necessary due to the fact that the user supplies the instruction set.

2.3 Stationary States

We will refer to "Stationary States" throughout this document. A Stationary State is a state in which all liveness requirements can be fulfilled. Both the state and the instruction history determine whether all of a process' liveness requirements *are* fulfilled; but a Stationary State only requires that a process' liveness requirements *can* be fulfilled. For example, imagine that a process has a single state variable "on" and a single liveness requirement "on $\sim>$ ring bell" (ring bell is some output event). Let the initial state be "on = $False$". The initial state is a Stationary State because all liveness requirements can be (and are) fullfilled. Notice that if the process sets "on = $True$" then "on = $False$" and never rings the bell then the process will be in a Stationary State but will not have fulfilled all of its liveness requirements.

Stationary States are important because they show an absense of a liveness requirement; any process in a stationary state cannot be trusted to make a move, and hence cannot be trusted to participate in helping to achieve a goal. This is particularly important in Peterson's Algorithm; if we make the initial state stationary then we can avoid synthesizing a round-robin style algorithm because no process can be trusted to cooperate. Instead, the synthesizer will be forced to find a solution where individual processes can enter the critical section without requiring the help of other processes.

2.4 Required States

The user may specify a subset of states that the program must visit. This may seem like a liveness requirement, but we do NOT consider it a liveness requirement here. Instead, we see this as a guide to help the synthesizer produce an algorithm that the user desires. In practice this means that we should not expect required states to be a motivator for processes to make progress. This is imortant for an algorithm like Peterson's where the intial state needs to be stationary, but the algorithm we desire needs to eventually enter the critical section.

2.5 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 power a user has to specify a program that they want and will find useful. The language will allow arbitrary safety requirements (in 1st order logic), but for the time being I will only allow a conjunction of "leads to" ($\sim>$) expressions for specifying liveness. In this document I will talk about the input as described by the input language, however, in practice I will hard code all inputs to avoid dealing with parsers and ASTs (at least for the first cut).

2.6 Instruction Set

The list of all possible instructions that the program can execute. This replaces the previous idea of a "transition constraint". I will hard code a function for each instruction for the time being.

3 Example

We will describe a set of requirements that should synthesize an algorithm that is close to Peterson's Mutual Exclusion Algorithm. 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. Required state: $\forall p \in ProcSet : cs[p]$. In other words, each process must be able to enter the critical section at some point.

4. Safety requirements:

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

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

5. Liveness requirements:

   (a) $\forall p \in ProcSet : flag[p] \ \sim> cs[p]$

   (b) $\forall p \in ProcSet : flag[p] \ \sim> \neg flag[p]$

   (c) $\forall p \in ProcSet : cs[p] \ \sim> \neg cs[p]$

6. Instruction Set: Approximately: flip $flag[p]$, flip $cs[p]$, flip $turn$, $turn = Other(p)$, $turn = p$, wait $turn[p]$, wait $flag[q]$, wait $turn[p] \wedge flag[q]$, wait $turn[p] \vee flag[q]$

Assuming the synthesizer produces a program without timing out, let's see what guarantees we can make for the following three properties [2]:

1. Satisfies Mutual Exclusion: given by the safety property (a)

2. Satisfies Progress: this is given by the safety property (a), the state requirement, and the fact that the initial state is a Stationary State. The fact that the initial state is stationary means that any process that enters the critical section cannot rely on any other process to help them; this prevents the synthesizer from producing a round-robin style algorithm. Safety property (a) implies that any process that wants to enter the critical section must assume that all other processes *may* also attempt to enter the critical section, and must cooperate with those processes to achieve its goal. The state requirement guarantees that the synthesizer will not stop working until it finds an algorithm that enters the critical state.

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 arbitrary and is not guaranteed to limit the number of times other processes can enter the critical section.

I may need to think more about how to express the Bounded Waiting requirement, but I don't necessarily see this as a must-do before implementation.

4 Implementation

Consider the following algorithm for synthesizing a program for p processes using a BFS:

```
input: requirements
queue = [noop]
while true:
  program = pop(queue)
  for instruction in AllInstructions:  // branching exp
    newProgram = program + instruction
    if satisfiesAllRequirements(newProgram, requirements):  // interleaving exp
      return newProgram
    if noSafetyViolations(newProgram, requirements):  // interleaving 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. Below I will talk about ideas for combating each exponential separately.

4.1 Efficiently Checking For Safety Violations

4.1.1 Probabilistic Analysis

To guarantee that a program satisfies all safety properties we must check every possible interleaving of instructions for all p processes. With a bit of help from the internet [3], we find that the number of possible interlreavings of n instructions with p processes is

$$f(n,p) = \frac{(pn)!}{(n!)^p}$$

Suppose we only want to check a single interleaving for a violation, let us calculate the probability of checking the correct one. There exists a finite trace τ that contains the safety violation, with the last state being the perpetrator [4]. If the safety violation was introduced at instruction i, then at least one of the $f(i,p)$ interleavings contain τ as a prefix; let's call this τ_i. At instruction j (where $j \geq i$) there are $f(j-i,p)$ interleavings that contain τ_i as a prefix. This implies that the probability that a random interleaving with j instructions finds the safety violation is:

$$\frac{f(j-i,p)}{f(j,p)}$$

Suppose, for a program with n instructions, we choose $g(n)$ interleavings independently and randomly to check for safety violations. Then the probability we find the safety violation τ *by the time* we have j instructions is at least:

$$1 - \prod_{k=i}^{j} \left(1 - \frac{f(k-i,p)}{f(k,p)}\right)^{g(k)}$$

Analytically, I was not able to figure out whether this probability is high or low. It was simple enough to test the results with a program though, and I discovered that the probability of finding the safety

violation by checking a polynomial number of interleavings (i.e. $g(n) = O(n^c)$, c is a constant) is dismally bad as i, the instruction where a violation is introduced, increases. This is not good news for the worst case scenario. However, it may be the case that when safety violations are introduced, they appear in many of the interleavings. Let us assume that the safety violation at instruction i is introduced to a fraction $c(i)$ of the interleavings, where $0 < c(i) < 1$. This means that there are $c(i) * f(i,p)$ interleavings with a safety violation at instruction i. This implies that the probability of finding a safety violation introduced at instruction i by the time we reach instruction j is:

$$1 - \prod_{k=i}^{j} \left(1 - \frac{c(i) * f(i,p) * f(k-i,p)}{f(k,p)}\right)^{g(k)}$$

Let's concentrate directly on the probability of finding the safety violation in a single interleaving. Notice that when the safety violation is first introduced, it is the case that $k = i$ and thus:

$$\frac{c(i) * f(i,p) * f(k-i,p)}{f(k,p)} = \frac{c(i) * f(i,p) * f(0,p)}{f(i,p)} = c(i)$$

Thus the probability of finding a safety violation when it is introduced can be expressed as the probability α:

$$\alpha = 1 - (1 - c(i))^{g(i)}$$

Notice that there is no product notation, and therefore no loop variable k, and therefore the function g accepts the instruction number i as its argument. We can solve for $g(i)$:

$$g(i) = \frac{\log(1-\alpha)}{\log(1-c(i))}$$

We clearly want α to be high, so we can fix its value around 0.9 or 0.99, and then $g(i)$ becomes a function of just $c(i)$. Then as $c(i)$ becomes a smaller and smaller fraction ($1e-10, 1e-11$, etc.) $g(i)$ becomes dominated by $\frac{-1}{\log(1-c(i))}$. In other words, we can determine $g(i)$ by choosing $c(i)$, which is our best guess for the fraction of interleavings that have safety violations when instruction i is introduced. This flexibility allows us to choose $g(i)$ potentially as a polynomial or even a constant which can greatly improve our runtime for checking safety violations.

Assuming that safety violations are introduced in batches that are some fraction $c(i)$ of the number of interleavings is a big assumption. However, if it is valid, it can drastically reduce the number of checks we need to perform while still yielding a high percent chance of catching safety violations. If the assumption is invalid the consequences are large: we pay an exponential price ($O(b^n)$) per branch that we *could* have pruned if we correctly detected the safety violation. Fortunately, missing safety violations only impacts performance; we can always do a full check of all interleavings for a program once it satisfies all requirements (I would likely need to use an SMT solver in practice) before returning it to the user as a correct solution.

4.2 Efficiently Searching For Programs

I have some ideas but they're not mature enough to add to the spec yet.

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`

[3] Stack Exchange.
https://math.stackexchange.com/questions/77721/number-of-instruction-interleaving

[4] Alpern, Shneider
https://www.cs.cornell.edu/fbs/publications/DefLiveness.pdf

[5] Alur et al. (2018) ACM SIGPLAN Notices, Volume 53, Issue 4
https://dl.acm.org/doi/pdf/10.1145/3296979.3192410