Abstraction-Guided Synthesis of Synchronization Master Seminar SS'2012

Sergey Grebenshchikov

Introduction

Applied in the implementation of parallel and distributed software, as well as a design pattern for non-parallel systems, concurrency is a powerful tool. Both application and verification of concurrent programs, however, pose additional difficulties not encountered in sequential programs.

One commonly required feature is that concurrent threads can read and write shared data. This presents a difficulty in design and verification: The state of shared data may depend on the order in which the operations of individual threads are scheduled. This results in additional state-space blow-up due to the multiplicity of possible thread interleavings, making it difficult (for humans) to reason about the behavior of concurrent programs and computationally hard (for algorithms) to analyse it.

The Abstraction-Guided Synthesis (AGS) algorithm we present below tackles the task of inferring synchronization primitives such that a given concurrent program satisfies a given safety specification.

1 The AGS algorithm

The AGS algorithm is a procedure that takes as input a (possibly unsafe) program P and a safety specification S and produces a modified program P' that satisfies the given specification. The success of the procedure is guaranteed if all serial (i.e. thread-atomic) executions of the progam satisfy the specification.

The core of the algorithm is the enumeration of counterexamples to S and *avoidance* of these counterexamples by program restriction. In the paper, the addition of *atomic sections* (thread sections in which context switches are prevented) is used as a mechanism to avoid counterexamples in the above sense.

To reduce the computational effort involved in enumerating program traces, and thus to make both analysis and synthesis more scalable, the AGS algorithm uses abstract interpretation, combined with abstraction refinement. AGS can thus be seen as a combination of a verifier and a synthesizer that enables a trade-off between verifier coarseness and synthesizer restrictiveness – if we accept more restrictive synchronization, we can reduce the computational effort by using a coarser abstraction.

Pseudocode for the AGS algorithm is shown in Algorithm 1. Some of the interesting subroutines, such as Avoid (generation of atomicity constraints) and Implement (generation of atomic sections in the output program) are discussed in greater detail below.

Algorithm 1 Abstraction-Guided Synthesis (Pseudocode)

```
**Input:** Program $P$, Specification $S$, Abstraction function $\alpha$.
**Output:** Program $P'$ satisfying $S$.

procedure AGS($P, S, \alpha$)
    $\varphi := \mathsf{T}$
    repeat
        if $\exists \pi \in \mathsf{AbstractTraces}(P|_\varphi, \alpha) : \pi \nvDash S$ then
            if $\mathsf{ShouldAvoid}(\pi, \alpha) \vee \neg\mathsf{CanRefine}(\alpha, \pi)$ then
                if $\mathsf{CanAvoid}(\pi)$ then
                    $\varphi := \varphi \wedge \mathsf{Avoid}(\pi)$
                else abort
            else
                $\alpha := \mathsf{Refine}(\alpha, \pi)$
        else
            return $\mathsf{Implement}(P|_\varphi)$
```

Algorithm 2 Avoid: Generation of atomicity constraints

```
procedure AVOID($\pi = \pi_1 \dots \pi_n$)

    $\rho := \bigvee_{i,j \in [n] \wedge i<j} \{[\pi_i, \pi_j] \mid \wedge \exists t : \mathrm{T}(\pi_i) = \mathrm{T}(\pi_j) = t \wedge \forall l \in ]i,j[ : \mathrm{T}(\pi_l) \neq t\}$

    return $\rho$
```

2 Minimality under abstraction

A restricted program $P' = \mathsf{Implement}(P|_\varphi)$ returned by AGS with respect to a fixed abstraction α and the safety specification S is *minimally-atomic* if any program obtained by removing or shrinking atomic sections no longer satisfies the safety specification under α (perhaps *irreducibly-synchronized w.r.t.* α *and* S would be a better term for this property). It is shown in the paper that for each minimally-atomic result program P', *there exists* a run of AGS yielding this program. This statement relies on *all* counterexample paths in each iteration being considered by AGS. Thus, to guarantee minimality, backtracking over the chosen counterexample paths is necessary.

To ensure minimality in the above sense in the presence of abstraction refinement, it is also necessary to discard the atomicity constraints φ after each refinement step. Minimality is then defined with respect to the abstraction used in the last iteration of AGS. If the atomicity constraints are not discarded, the resulting program is not necessarily minimally-atomic.

3 Encoding of trace infeasibility

To avoid a counterexample trace π by adding atomic sections, AGS generates an *atomicity constraint* to encode the infeasibility of π in the output program. The constraint φ computed by the procedure Avoid. We examine this process on a small example. Consider the program shown in Fig. 1 together with the safety specification that the assertion in thread T must not be violated (doing so leads to an error state). Under parity abstraction over the variables y_1 and y_2 and identity abstraction over x and z, we obtain the counterexample trace $\pi_1 = S_1 R_1 T_1 S_2 R_2 T_2 T_3$ (we

```
R() {          S() {          T() {
   x+=z;          z++;           y1 = f(x);
   x+=z;          z++;           y2 = x;
                                 assert( y1 ≠ y2 );
}              }              }
                              f(x) {
                                 if(x == 1)
                                    return 3;
                                 else if(x == 2)
                                    return 6;
                                 else return 5;
                              }
```

Figure 1: Example: $R \parallel S \parallel T$, f atomic

use the notation T_i to refer to the i-th statement of thread T). To avoid this trace, we compute the (disjunction of) atomicity constraints $\mathsf{Avoid}(\pi_1)$ by enumerating the context switches within π_1 and requiring at least one of them to be disabled in the output program, thus making π_1 infeasible. The trace π_1 has the three context switches $S_1 \rightarrow R_1T_1 \rightarrow S_2$, $R_1 \rightarrow T_1S_2 \rightarrow R_2$ and $T_1 \rightarrow S_2R_2 \rightarrow T_2$. We thus obtain the atomicity constraint

$$\mathsf{Avoid}(\pi_1) = [S_1, S_2] \vee [R_1, R_2] \vee [T_1, T_2] =: \psi_1$$

Here, the notation $[S_1, S_2]$ represents the constraint that no context switch can occur between the statements S_1 and S_2. Restricting the set of abstract traces with this constraint and keeping the parity abstraction function, we obtain an abstract reachability tree (ART) containing a counterexample path $\pi_2 = R_1S_1S_2R_2T_1T_2T_3$ (note that this path satisfies, as it must, the atomicity constraint φ_1: the statements S_1 and S_2 are executed atomically). The path π_2 has only one context switch, namely $R_1 \rightarrow S_1S_2 \rightarrow R_2$, thus we obtain the atomicity constraint

$$\mathsf{Avoid}(\pi_2) = [R_1, R_2] =: \psi_2$$

Restricting the set of abstract traces further by ψ_2, one further counterexample $\pi_3 = S_1R_1R_2S_2T_1T_2T_3$ is reachable, yielding the atomicity constraint $\psi_3 = [S_2, S_2]$. After restiction with ψ_3, error states are no longer present in the ART and the algorithm terminates. During this run of AGS, we have accumulated the atomicity constraint

$$\varphi = \psi_1 \wedge \psi_2 \wedge \psi_3 = ([S_1, S_2] \vee [R_1, R_2] \vee [T_1, T_2]) \wedge [R_1, R_2] \wedge [S_1, S_2]$$

A minimal satisfying assignment is $[R_1, R_2] \wedge [S_1, S_2]$, resulting in the atomic sections shown in the output program in Fig. 2. The implemented atomic sections ensure that the program satisfies the safety specification and suffice to prove this under parity abstraction.

4 Atomicity constraints as SAT instances

If a singleton atomicity constraint $[a, b]$ is represented as a propositional variable $X_{[a,b]}$, the composite atomicity constraint $\varphi = \varphi_i \wedge \ldots \wedge \varphi_n$ with $\varphi_i = [a_{i1}, b_{i1}] \vee \ldots \vee [a_{ik_i}, b_{ik_i}]$ could be represented by replacing $[a_{i1}, b_{i1}]$ by $X_{[a,b]}$ in each φ_i to obtain a propositional formula in positive CNF that can be passed to a SAT solver. Denoting

```
R() {
  atomically {
    x+=z;
    x+=z; }
}
```

```
S() {
  atomically {
    z++;
    z++; }
}
```

```
T() {
  y1 = f(x);
  y2 = x;
  assert( y1 ≠ y2 );
}
f(x) {
  if(x == 1)
    return 3;
  else if(x == 2)
    return 6;
  else return 5;
}
```

Figure 2: Example: $\mathsf{AGS}(R \parallel S \parallel T)$, f atomic

the immediate dominator of a by $\mathsf{IDom}(a)$ and the immediate postdominator of b by $\mathsf{IPDom}(b)$, the atomic section represented by an assingment of true to $X_{[a,b]}$ begins at $\mathsf{IDom}(a)$ and ends at $\mathsf{IPDom}(b)$. Overlapping atomic sections are joined to begin at the common immediate dominator and end at the common postdominator.

This way of choosing atomic sections is aimed to maximize the number of counterexamples eliminated by a single atomic section and thus minimize the total number of atomic sections. It however, wholly ignores the size of the sections, which may well have a considerable impact on the number of valid traces that are eliminated along with the erroneous ones.

To obtain smaller few atomic regions, a singleton atomicity constraint $[a,b]$ for a thread T could be represented a propositional formula over the variables T_i corresponding to program locations of the thread T. We represent the atomic section beginning at $\mathsf{IDom}(a)$ and ending at $\mathsf{IPDom}(b)$ by the formula $F([a,b])$:

$$F([a,b]) = \bigwedge \{T_i \in T : T_i \text{ dominated by } \mathsf{IDom}(a) \wedge \text{post-dominated by } \mathsf{IPDom}(b)\}$$

A composite atomicity constraint

$$\varphi = \varphi_1 \wedge \varphi_2 \ldots \wedge \varphi_n$$

where

$$\varphi_i = [a_{i1}, b_{i1}] \vee \ldots \vee [a_{ik_i}, b_{ik_i}]$$

can then be represented by the propositional logic formula

$$F(\varphi) = F(\varphi_1) \wedge \ldots \wedge F(\varphi_n)$$

where

$$F(\varphi_i) = F([a_{i1}, b_{i1}]) \vee \ldots \vee F([a_{ik_i}, b_{ik_i}])$$

A minimal satisfying assignment for $F(\varphi)$ corresponds to a choice of atomic sections that maximizes the number of counterexamples eliminated by a single atomic section, but also minimizes the size of these sections.

5 Limitations

In the presence of synchonization primitives in the input program, AGS does not necessarily preserve termination of the input program. Consider the following example, where b and c are semaphores, initially both at a value of 0:

```
S {down(b);                          T { ⎡up(b);
   ⎡up(c);                               ⎣down(c);
   ⎣down(b);  }                          ⎡up(b);
                                         ⎣assert(b == 1);  }
```

Figure 3: Example program P: introduction of deadlocks by AGS. Alternative atomic sections proposed by AGS are shown as brackets around the corresponding statements.

The primitives up and down behave in the standard manner: up atomically increments the semaphore; down blocks while the semaphore is at its lower bound and atomically decrements it once it is not.

The blocking behavior of down for two threads is correctly modeled by enabling the corresponding transition if the semaphore is not at 0 and disabling it otherwise. We obtain the following concrete (α = id) reachability tree for $P = S \parallel T$:

$$\begin{tikzcd}
& (0,0,\ 0,0) \arrow[d, "T_1"] & \\
& (0,1,\ 1,0) \arrow[d, "S_1"] & \\
& (1,1,\ 0,0) \arrow[d, "S_2"] & \\
& (2,1,\ 0,1) \arrow[d, "T_2"] & \\
& (2,2,\ 0,0) \arrow[d, "T_3"] & \\
& (2,3,\ 1,0) \arrow[dl, "S_3"'] \arrow[dr, "T_4"] & \\
(3,3,\ 0,0) \arrow[d, "T_4 \lightning"'] & & (2,4,\ 1,0) \arrow[d, "S_3"] \\
\text{error} & & (3,4,\ 0,0)
\end{tikzcd}$$

P

$$\begin{tikzcd}
(0,0,\ 0,0) \arrow[d, "T_1"] \\
(0,1,\ 1,0) \arrow[loop below] \\
\text{T blocked inside atomic section}
\end{tikzcd}$$

$P|_{[T_1,T_2]}$

$$\begin{tikzcd}
(0,0,\ 0,0) \arrow[d, "T_1"] \\
(0,1,\ 1,0) \arrow[d, "S_1"] \\
(1,1,\ 0,0) \arrow[d, "S_2"] \\
(2,1,\ 0,\ 1) \arrow[loop below] \\
\text{S blocked inside atomic section}
\end{tikzcd}$$

$P|_{[S_2,S_3]}$

$$\begin{tikzcd}
(0,0,\ 0,0) \arrow[d, "T_1"] & \\
(0,1,\ 1,0) \arrow[d, "S_1"] & \\
(1,1,\ 0,0) \arrow[d, "S_2"] & \\
(2,1,\ 0,1) \arrow[d, "T_2"] & \\
(2,2,\ 0,0) \arrow[d, "T_3"] & \\
(2,3,\ 1,0) \arrow[dr, "T_4"] & \\
& (2,4,\ 1,0) \arrow[d, "S_3"] \\
& (3,4,\ 0,0)
\end{tikzcd}$$

$P|_{[T_3,T_4]} = P|_{\varphi}$

Figure 4: Concrete reachability trees for P and its restrictions. The counterexample path $\pi = T_1 S_1 S_2 T_2 T_3 S_3 T_4$ is emphasized. States are represented as tuples $(\mathrm{pc}_S, \mathrm{pc}_T, b, c)$ of program variable valuations.

There is exactly one path, namely $\pi = T_1 S_1 S_2 T_2 T_3 S_3 T_4$, that violates the assertion. The path contains three context switches, namely $T_1 \to S_1 S_2 \to T_2$, $S_2 \to T_2 T_3 \to S_3$ and $T_3 \to S_3 \to T_4$. Thus we obtain

$$\mathsf{Avoid}(\pi) = [T_1, T_2] \vee [S_2, T_3] \vee [T_3, T_4] =: \varphi$$

The reachability trees for the restrictions of P to each of the disjuncts of φ are shown in Fig. 4. Since all traces in $\mathsf{Traces}(P|_{[T_1,T_2]}) \cup \mathsf{Traces}(P|_{[S_2,S_3]})$ are prefixes of traces in $\mathsf{Traces}(P|_{[T_3,T_4]})$, the reachability tree of $P|_\varphi$ is equal to that of $P|_{[T_3,T_4]}$.

All disjuncts of φ ensure safety: there are no runs violating the assertion in the corresponding reachability trees. Each disjunct represents a minimal solution, thus the choice made by AGS is arbitrary.

While the restriction with $[T_3, T_4]$ eliminates only the counterexample path and preserves termination of the input program, the restriction with either $[S_2, S_3]$ or $[T_1, T_2]$ also avoids the assertion violation, but introduces a deadlock, in which a thread is blocked in its atomic section, waiting for an $\mathsf{up}(.)$ operation of the other thread.

If the arbitrary choice between the three minimal solutions yields either $[S_2, S_3]$ or $[T_1, T_2]$, AGS introduces a deadlock that could be avoided by a different synchronization available to the algorithm.