An Overview of Previous Work on Verification of Distributed Systems

Steven Schaefer

University of Michigan

February 3rd, 2023

Finding Inductive Invariants

Given a distributed system as a transition system $(\mathcal{S}, \mathcal{I}, \mathcal{T})$ and a desired property P.
Want to prove safety property P true for all executions of a system. However, a property P is not in general *inductive* — preserved by the transition relation — so we need to find *inductive invariants* I_j such that $\bigwedge I_j \rightarrow P$.
Inductive invariants can be found using tools like I4, IC3PO, etc.

Describing the Reachable States

In addition to finding inductive invariants that imply the desired property, we can also describe the reachable states R of the system using inductive invariants. We may find enough inductive invariants I_j such that their conjunction describes R exactly.

$$\bigcap I_j = R$$

Algorithm to Find R

Note: We will walk through an example.

1. Instantiate the protocol for small, finite sizes.
2. Enumerate all reachable states for each instantiation.
3. Perform logic minimization on the reachable states.
4. Infer a quantified formula for each clause orbit.
5. If the formula is syntactically the same as the previous iteration, stop. Otherwise, repeat step 1 with a larger instantiation.

Client-Server Example

Sorts: $\texttt{client}, \texttt{server}$. States: $\mathcal{S} = \{\texttt{link}(c, s), \texttt{semaphore}(s)\}$.
Initial State: $\mathcal{I} = (\forall c, s.\neg\texttt{link}(c, s)) \wedge (\forall s.\texttt{semaphore}(s))$
Transitions: `connect` and `disconnect`.

```
connect(c, s) {
    require semaphore(s);
    link(c, s) := true;
    semaphore(s) := false;
}

disconnect(c, s) {
    require link(c, s);
    link(c, s) := false;
    semaphore(s) := true;
}
```

Safety Property: $\forall c_1, c_2.\forall s.\neg\texttt{link}(c_1, s) \vee \neg\texttt{link}(c_2, s) \vee c_1 = c_2$

Client-Server Example: Generating Reachable States

Choose finite instantiation $|\texttt{client}| = 2, |\texttt{server}| = 2$.

Unminimized R:

$\texttt{l}(c_1, s_1)$	$\texttt{l}(c_1, s_2)$	$\texttt{l}(c_2, s_1)$	$\texttt{l}(c_2, s_2)$	$\texttt{sem}(s_1)$	$\texttt{sem}(s_2)$
0	0	0	0	1	1
1	0	0	0	0	1
0	1	0	0	0	1
0	0	1	0	0	1
0	0	0	1	1	0
1	0	0	1	0	0
0	0	1	1	0	0
0	1	1	0	0	0
1	1	0	0	0	0

Read top row as $\neg\texttt{link}(c_1, s_1) \wedge \neg\texttt{link}(c_1, s_2) \wedge \neg\texttt{link}(c_2, s_1) \wedge \neg\texttt{link}(c_2, s_2) \wedge \texttt{semaphore}(s_1) \wedge \texttt{semaphore}(s_2)$.

An Aside on Logic Minimization

Idea: find a formula that is logically equivalent to the original formula, but of minimal size.
Simple example: $\phi := AB + A'B = (A + A')B = B$. Represented as a table,

Unminimized

A	B	ϕ
0	0	0
0	1	1
1	0	0
1	1	1

Minimized

A	B	ϕ
-	0	0
-	1	1
-	0	0
-	1	1

Quine-McCluskey algorithm repeatedly apply this idea until a minimal formula is found. *Note:* for those who know about Gröbner bases, this is a special case of the Buchberger algorithm for Boolean algebras.
Can also use heuristic tools like the Espresso minimizer.

Client-Server Example: Minimizing Reachable States

Negate the reachable states and minimize the resulting formula.
Note: each entry in this table represents an element of $\neg R \subset \mathcal{S}$.

Minimized $\neg R$:

$\mathtt{l}(c_1, s_1)$	$\mathtt{l}(c_1, s_2)$	$\mathtt{l}(c_2, s_1)$	$\mathtt{l}(c_2, s_2)$	$\mathtt{sem}(s_1)$	$\mathtt{sem}(s_2)$
1	-	1	-	-	-
-	1	-	1	-	-
1	-	-	-	1	-
-	-	1	-	1	-
-	1	-	-	-	1
-	-	-	1	-	1
0	-	0	-	0	-
-	0	-	0	-	0

Read top row as $\mathtt{link}(c_1, s_1) \wedge \mathtt{link}(c_2, s_1)$. Interpret as saying this is a bad state.

Client-Server Example: Minimizing Reachable States

Negate again to get statements about good states — i.e. describing R.

Minimized $\neg R$:

$\texttt{l}(c_1, s_1)$	$\texttt{l}(c_1, s_2)$	$\texttt{l}(c_2, s_1)$	$\texttt{l}(c_2, s_2)$	$\texttt{sem}(s_1)$	$\texttt{sem}(s_2)$
1	-	1	-	-	-
-	1	-	1	-	-
1	-	-	-	1	-
-	-	1	-	1	-
-	1	-	-	-	1
-	-	-	1	-	1
0	-	0	-	0	-
-	0	-	0	-	0

Instead of reading top row as stating a bad state, after negation we find the top row states that a good state must obey $\neg\texttt{link}(c_1, s_1) \vee \neg\texttt{link}(c_2, s_1)$.

Client-Server Example: Infer Quantifiers

Minimized $\neg R$:

$\texttt{l}(c_1, s_1)$	$\texttt{l}(c_1, s_2)$	$\texttt{l}(c_2, s_1)$	$\texttt{l}(c_2, s_2)$	$\texttt{sem}(s_1)$	$\texttt{sem}(s_2)$
1	-	1	-	-	-
-	1	-	1	-	-
1	-	-	-	1	-
-	-	1	-	1	-
-	1	-	-	-	1
-	-	-	1	-	1
0	-	0	-	0	-
-	0	-	0	-	0

For each clause orbit — rows 1 and 2 are symmetric — we infer a quantifed formula. Resulting in the following formulas:

- $\forall s.\exists c.\texttt{link}(c, s) \vee \texttt{semaphore}(s)$
- $\forall c.\forall s.\neg\texttt{link}(c, s) \vee \neg\texttt{semaphore}(s)$
- $\forall c_1, c_2.\forall s.\neg\texttt{link}(c_1, s) \vee \neg\texttt{link}(c_2, s)$ — almost what we want, but doesn't generalize to larger sizes

Client-Server Example: Where Generalization Fails

From the last slide, we found $\forall c_1, c_2. \forall s. \neg \texttt{link}(c_1, s) \vee \neg \texttt{link}(c_2, s)$. But recall the property we want to prove is, $\forall c_1, c_2. \forall s. \neg \texttt{link}(c_1, s) \vee \neg \texttt{link}(c_2, s) \vee c_1 = c_2$. To observe this mutual exclusion property we need at least 3 clients, which is the cutoff here. Need enough clients to distinguish between "for all clients but one" and "one out of every pair of clients".

$|\texttt{client}| = 3, |\texttt{server}| = 2$ gives a formula that generalizes to any number of clients and servers.

Client-Server Example: Formula for R

- $I_1 = \forall s.\exists c.\texttt{link}(c, s) \vee \texttt{semaphore}(s)$
- $I_2 = \forall c.\forall s.\neg\texttt{link}(c, s) \vee \neg\texttt{semaphore}(s)$
- $I_3 = \forall c_1, c_2.\forall s.\neg\texttt{link}(c_1, s) \vee \neg\texttt{link}(c_2, s) \vee c_1 = c_2$

$R = I_1 \wedge I_2 \wedge I_3$, and $P = I_3$. These formulas hold for any number of clients and servers, shown with Z3.

Good Minimization

The process we just witnessed requires minimization to be good.
Initially, we used Espresso which is heuristic and not guaranteed to find the smallest formula. Additionally, Espresso — and other logic minimization methods — do not a priori give symmetric solutions.
Need symmetry to be able to infer quantifiers.
Solution: encode symmetric minimization into SAT, then use a SAT solver to find a symmetric and minimal solution.

Client-Server Example: A Disjunctive View

Idea: find a small number of formulas that cover the reachable states. Then, use a disjunction to combine them.
The output of this process is another expression for R which may give different semantic insight than the previous process.
Effectively, this gives an abstract transition system that simulates the original system. The benefit is that this abstract system has a much smaller state space, and its size is independent of sort sizes.
TODO: include some pictures. Note that this process was largely manual and that the resulting formulas, while correct, more or less just take the previous process and distribute it over some counting abstraction scheme.

Future Work in This Area

- The process stops when the outputted formula is syntactically equivalent across several instantiation sizes. There is no proof that this is sufficient to guarantee that the formula is correct for all sizes. We witness that it is sufficient, and verify the equivalence manually, but a proof would be great.
- Some quantifer inference is ambiguous. Which is especially apparent at small sizes. Need to find a way to resolve this ambiguity. Example: When in size 3 and you witness a behavior involving 2 nodes, is this supposed to be thought of as an "all-but-one" behavior or a "every-pair" behavior? Quantifier inference needs a more rigorous treatment — what is inferable, the normal forms of the resulting formulas, etc.
- Automatic inference of cardinality of R. In the client-server example, it can be shown that $|R| = (|\texttt{client}| + 1)^{|\texttt{server}|}$. Observable empirically, and manually provable after a little thought, but would be nice to extract automatically.
- What sorts of abstraction schemes can be used to make the disjunctive view more tractable?

Current Project

The Ivy language is used for reasoning about distributed systems. Elanor and I are working on a project investigating Ivy in more detail. We are interested in the following questions:

- How can the semantics of Ivy be formalized?
- How can we reason about the correctness of existing proofs about Ivy programs?
- Given a protocol written in Ivy, can we output an implementation of that protocol that is correct by construction, say in C/C++?