Compositional Reasoning and Learning for Model Generation

Sanjit A. Seshia
EECS, UC Berkeley

Acknowledgments: Avrim Blum

Compositional Reasoning
Need for Compositional Reasoning

- Model checking “flat” designs/programs does not scale
  - Can be applied locally, to small modules
  - Globally to simplified models
- Model checking simplified, flat designs is mainly a “best-effort debugging” tool

How do we scale up the method so we can use it for “verification”, not just “debugging”? 

Compositional Reasoning: Divide-and-Conquer

- Idea: use proof techniques to reduce a property to easier, localized properties.

\[
\text{property} \quad \xrightarrow{\text{decomposition}} \quad \text{abstraction} \quad \xrightarrow{\text{proof assistant}} \quad \text{verification} \quad \xrightarrow{\text{model checker/decision procedure}}
\]
Notation

Proof rule specified as:

\[ A_1 A_2 A_3 \ldots A_n \quad C \]

assumptions

conclusion

Assume/Guarantee Reasoning

• System and its Environment

• Each makes an assumption about the other’s behavior
• In return, each guarantees something about its own behavior

• Come up with a proof rule
  – Assumptions are what we verify
  – Conclusion is the desired property
Simple assume/guarantee proof

- Thus, we localize the verification process
- Note abstraction is needed to benefit from decomposition (why?)

Mutual property dependence

- What about the case of mutual dependence?
- Note, this doesn’t work (why?)
“Circular” compositional proofs

• Let \( p \rightarrow q \) stand for
  “if \( p \) up to time \( t-1 \), then \( q \) at \( t \)”

• Equivalent in LTL of
  \( \neg(p \ U \neg q) \)

• Now we can reason as follows:

\[
\begin{align*}
q \rightarrow p & \quad \text{verify using } A \\
p \rightarrow q & \quad \text{verify using } B
\end{align*}
\]

\( Gp \wedge Gq \)

That is, \( A \) only has to “behave” as long as \( B \) does, and vice-versa.

Model Generation

• Generating models of finite-state systems by observing execution traces
  – Based on a machine learning algorithm first proposed by D. Angluin in ’87 and improved upon by Rivest & Schapire in ’93
Setting

State variables $V = V_E \cup V_M, V_E \cap V_M = \emptyset$

Want to observe $E$ and generate a good model of it
Usually easy to get a model of $M$

Why Learn Models?

Generating Environment models

- As a middle ground between
  - Traditional, pessimistic (worst-case) verification
  - Optimistic verification (“does there exist an environment that makes my system work?”)

- To generate environment assumptions for use in assume-guarantee reasoning

Another use: Generating Abstractions
A Quote

• “Assumptions are the things you don't know you're making”
  — Douglas Adams, Mark Cawardine, "Last Chance to See"

Learning Env. Model

• Model: (Deterministic) Finite Automaton
  – As a representation of the set of traces of env.

• What we can do:
  – Provide inputs to the environment
  – Observe (finite) prefixes of environment’s output trace

• Note:
  – Env. is a reactive system too, has infinitely long traces but we can only observe finite prefixes
  – So we are learning a finite automaton (not a Buchi automaton)
An Intuitive View

- Environment is a box, with input buttons and output lights
  - Outputs capture observable part of env state
- We can press some subset of input buttons at any time step
- Observe what lights turn on

Assumption for this lecture:
We can “reset” the environment at any time

Angluin’s DFA Learning Algo.
(adapted to our setting)

- Input: A box as in the previous picture
  - inputs from an alphabet $\Sigma$
- Outputs: a DFA that accurately represents all (finite) output traces seen so far
- What it can do:
  - Generate environment traces by supplying inputs
  - Ask an oracle whether a candidate DFA is indeed correct (if not, get a counterexample)
  - Reset environment model to initial state
Angluin’s DFA Learning Algo.
(adapted to our setting)

• Input: A box as in the previous picture
  • inputs from an alphabet $\Sigma$

• Outputs: a DFA that accurately represents
  all (finite) output traces seen so far
  • Given an oracle that precisely knows the
    environment, it learns the DFA whose language is
    exactly the output traces of the env.

• What it can do:
  – Generate environment traces by supplying
    inputs
  – Ask an oracle whether a candidate DFA is
    indeed correct (if not, get a counterexample)
  – Reset environment model to initial state

Formal Setup

• Want to learn (synthesize) a DFA $(Q, \Sigma, \delta, L)$
  – $Q$ : set of states
  – $\Sigma$ : input alphabet
  – $\delta$ : transition function: $Q \times \Sigma \rightarrow Q$
  – $L$ : labeling/output function

• What does it mean for two states of the
  DFA to be different?
  (In terms of the labels we observe)
Formal Setup

• Want to learn a DFA \((Q, \Sigma, \delta, L)\)
  – \(Q\) : set of states
  – \(\Sigma\) : input alphabet
  – \(\delta\) : transition function: \(Q \times \Sigma \rightarrow Q\)
  – \(L\) : labeling/output function

• What does it mean for two states of the DFA to be different?
  – \(q\) and \(q'\) are different if there is a input sequence s.t. the states reachable on that sequence from \(q\) and \(q'\) respectively have different labels

What defines a state

• Its label (observable part)
• The input sequence that reaches that state
  – Could be many, pick a representative
• The output sequences we see from that state
  – Perform “experiments” from that state to see this

• Angluin’s algorithm “names” a state by the latter two things
  – A prefix and a suffix
Algorithm Sketch

1. Start with only the DFA's initial state $q_0$
2. Generate a “new” state by supplying inputs
3. Check if its next states are observationally different from those of existing states
   - If yes, add it in
   - If not, ask the oracle if we have the correct DFA
     • If yes, we’re done
     • If not, use the counterexample to figure out what new state(s) to add so that counterex goes away
   - Go back to step 2

An Example

This is the DFA we want to learn (the correct environment model)
How the algorithm works on the previous example – worked out on board

Complexity

- Polynomial in size of environment model
- Good if environment model is small
  - This is why it is especially good for learning assumptions or concise env specifications
Some Early Refs.

• “Adaptive Model Checking” -- Groce, Peled, Yannakakis, TACAS’02
• “Learning Assumptions for Compositional Verification” -- Cobleigh et al., TACAS’03