Today’s Lecture

- **Symmetry Reduction**
  - Group states into equivalence classes by exploiting symmetries in the model
- **Compositional Reasoning**
  - Exploiting modularity by “assume-guarantee” reasoning
- **Mu-calculus & the “Property Hierarchy”**
Symmetry

- Many systems have inherent symmetry
  - Overall system might be composed of $k$ identical modules
  - E.g., a multi-processor system with $k$ processors
  - E.g., a multi-threaded program with $k$ threads executing the same code with same inputs
  - Anything with replicated structure

- Question: How can we detect and exploit the symmetry in the underlying state space for model checking?

Symmetry in Behavior

- Given a system with two identical modules
  - Run: $s_0$, $s_1$, $s_2$, ...
  - Trace: $L(s_0)$, $L(s_1)$, $L(s_2)$, ...

  - Each $s_i = (s_{i1}, s_{i2}, \text{rest})$ comprises values to variables of both modules 1 and 2
  - If we can interchange these without changing the set of traces of the overall system, then there is symmetry in the system behavior
Exploiting Symmetry

- If a state space is symmetric, we can group states into equivalence classes
  - Just as in abstraction

- Resulting state graph/space is called “quotient” graph/space
  - Model check this quotient graph

Quotient (first attempt)

\[ M = (S, S_0, R, L) \]

Let \( \cong \) be an equivalence relation on \( S \)

Assume: \( s \cong t \iff L(s) = L(t) \)

& \( s \in S_0 \iff t \in S_0 \)

Quotient: \( M' = (S', S'_0, R', L') \)

- \( S' = S/\cong \), \( S'_0 = S_0/\cong \) (states are equivalence classes with respect to \( \cong \))
- \( R'([s], [t]) \) whenever \( R(s, t) \)
- \( L'([s]) = L(s) \)
Is that definition enough?

Suppose we want to check an invariant:
Does M satisfy $\varphi$?

Instead if we check:
Does quotient M’ satisfy $\varphi$?

If M’ is constructed using the definition of $\equiv$ on the previous slide, will the above check generate spurious counterexamples?

Stable Equivalences

Equivalence $\equiv$ is called stable if:
$R(x, y) \Rightarrow$
for every s in [x]
there exists some t in [y] such that $R(s, t)$

Claim: Suppose $\equiv$ is stable, then:
$M$ satisfies $\varphi$ iff $M’$ satisfies $\varphi$
(Why?)
Detecting Symmetry

• Given symmetry expressed as an equivalence relation between states, we know how to exploit it
• How do we detect/compute this equivalence relation?
  – Need to characterize it more formally

Symmetry as Permutation

• Symmetry in the state space can be viewed as “equivalence under permutation”
• Permute the set of states so that the set of traces remains the same
  – A subset of states that remains the same under permutation forms the needed equivalence class
• A representation of all possible such permutations represents symmetry in the system
A permutation function $f : S \rightarrow S$ is an automorphism if:

$$R(s, t) \iff R(f(s), f(t))$$

What is an example automorphism for this state space?

- $f$: $f(0,0) = 1,1$, $f(1,1) = 0,0$
  - $f(0,1) = 0,1$, $f(1,0) = 1,0$

- $g$: $g(0,0) = 0,0$, $g(1,1) = 1,1$
  - $g(0,1) = 1,0$, $g(1,0) = 0,1$

$A = \{ f, g, f \circ g, \text{id} \}$

The set of all automorphisms forms a group!
Equivalence using Automorphisms

Let \( s \cong t \)
if there is some automorphism \( f \) such that
\( f(s) = t \) (and \( L(s) = L(t) \land s \in S_0 \iff t \in S_0 \))

The equivalence classes of an automorphism
(sets mapped to themselves) are called orbits

Claim 1: \( \cong \) is an equivalence
Claim 2: \( \cong \) is stable \( \text{ (why?)} \)

Orbits

\[
\begin{align*}
\{ (0,0), (1,1) \} \\
\{ (0,1), (1,0) \}
\end{align*}
\]
Symmetry reduction

Map each state to its representative in the orbit

How Symmetry Reduction works in practice

- A permutation (automorphism) group is manually constructed
  - Syntactically specify which modules are identical
- Orbit relation (equivalence relation) automatically generated from this
  - Using fixpoint computation (MC, Sec. 14.3)
- An (lexicographically smallest) element of each equivalence class is picked as its representative
- $S_0'$ and $R'$ generated from orbit relation
- Model checking explores only representative states
Symmetry reduction

- Implemented in many model checkers
  - E.g., SMV, Murϕ (finite-state systems), Brutus (security protocols)

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.

property \rightarrow \text{decomposition} \rightarrow \text{abstraction} \rightarrow \text{verification}

\{ \text{proof assistant} \}
\{ \text{model checker/decision procedure} \}
Notation

Proof rule specified as:

\[ A_1, A_2, A_3, \ldots, A_n \quad \text{assumptions} \]

\[ \quad C \quad \text{conclusion} \]

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

\[ p \rightarrow q \]

- 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?

\[ p \Rightarrow q \]

- 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
  \( \neg (p \mathcal{U} \neg q) \)
• Now we can reason as follows:

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

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

Temporal case splitting

Idea:
Split cases on most recent writer \( w \) at time \( t \).

\[
\phi \land \forall i: G((w=i) \Rightarrow \phi) \Rightarrow G\phi
\]

Rule can be used to focus within large process arrays... but still need to deal with interdependencies
Combine with circular reasoning

To prove case $w = i$ at time $t$, assume general case up to $t-1$:

$$\phi \land \forall i: G(\phi \Rightarrow ((w = i) \Rightarrow X\phi))$$

$$G\phi$$

still have many cases to prove...

Reduction by symmetry

By symmetry, suffices to prove that writes by $p_1$ are O.K.:

$$\phi \land G(\phi \Rightarrow ((w = 1) \Rightarrow X\phi))$$

verify using $p_1$

$$G\phi$$
The Mu-Calculus

Property Hierarchy

Mu Calculus

CTL*

CTL

Buchi automata

LTL

LTL without X

Legend:  C
The Mu-Calculus

A recursive language for writing symbolic model-checking algorithms

\[ \text{EF } a = \mu Z (a \lor \text{EX } Z) \]
\[ \text{AG } a = \nu Z (a \land \text{AX } Z) \]

Mu-Calculus Syntax

\[ \phi ::= a \mid \neg a \mid Z \mid \phi \land \psi \mid \phi \lor \psi \mid \text{EX } \phi \mid \text{AX } \phi \mid \mu Z \phi \mid \nu Z \phi \mid Z : \text{region variable} \]

Any predicate transformer thus expressed is monotonic, hence all fixed points exist
Mu-Calculus Semantics

\[
[[ a ]]_{\text{Env}} := \langle a \rangle \\
[[ \neg a ]]_{\text{Env}} := \Sigma \setminus \langle a \rangle \\
[[ \varphi \land \psi ]]_{\text{Env}} := [[ \varphi ]]_{\text{Env}} \cap [[ \psi ]]_{\text{Env}} \\
[[ \varphi \lor \psi ]]_{\text{Env}} := [[ \varphi ]]_{\text{Env}} \cup [[ \psi ]]_{\text{Env}} \\
[[ \text{EX} \varphi ]]_{\text{Env}} := \text{pre}( [[ \varphi ]]_{\text{Env}} ) \\
[[ \text{AX} \varphi ]]_{\text{Env}} := \forall \text{pre}( [[ \varphi ]]_{\text{Env}} )
\]

Env maps each region variable to a region
\( \Sigma \) is the universe
pre and \( \forall \text{pre} \) compute set of previous states

Operational Semantics of Mu-Calculus

\[
[[ \mu Z \varphi ]]_E := S' := \emptyset; \\
\text{repeat } S := S'; S' := [[ \varphi ]]_{E(Z \rightarrow S)} \text{ until } S' = S; \\
\text{return } S
\]

\[
[[ \nu Z \varphi ]]_E := S' := \Sigma; \\
\text{repeat } S := S'; S' := [[ \varphi ]]_{E(Z \rightarrow S)} \text{ until } S' = S; \\
\text{return } S
\]

Model checking works as above
Complexity

- Every $\mu/\nu$ alternation adds expressiveness
- Buchi automata in alternation depth of 2
- Model checking complexity:
  $O\left( (|\varphi| \cdot N)^d \right)$
  for formulas of alternation depth $d$
  - $N$ is size of model
- Most common implementation (SMV, Mocha):
  use BDDs to represent Boolean regions

Next class

- Model checking pushdown systems
  – Finite state control with a stack