EECS 219C: Computer-Aided Verification

Symmetry Reduction, Compositional Reasoning, Mu-Calculus

Sanjit A. Seshia
EECS, UC Berkeley

Acknowledgments:
T. Henzinger, K. McMillan, S. Rajamani

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?

S. A. Seshia 3

Symmetry in Behavior

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

S. A. Seshia 4

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 φ ?

Instead if we check:

Does quotient M' satisfy φ ?

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

Stable Equivalences

Equivalence $\cong$ 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 $\cong$ is stable, then:

M satisfies φ iff M' satisfies φ

(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

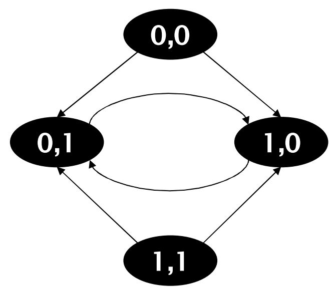
Automorphisms

A permutation function

$f : S \rightarrow S$

is an automorphism if:

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



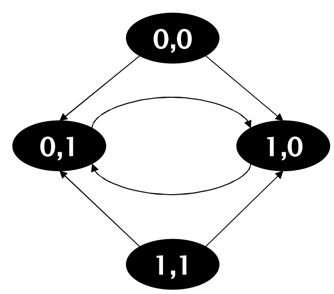
What is an example automorphism for this state space?

Automorphisms

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

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

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



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 $L(s) = L(t) \wedge s \in S_0$ iff $t \in S_0$)

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

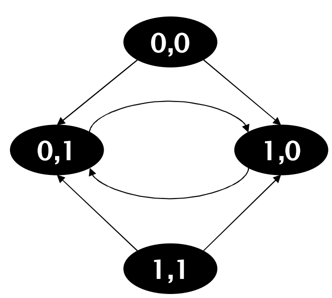
Claim 1: $\cong$ is an equivalence

Claim 2: $\cong$ is stable (why?)

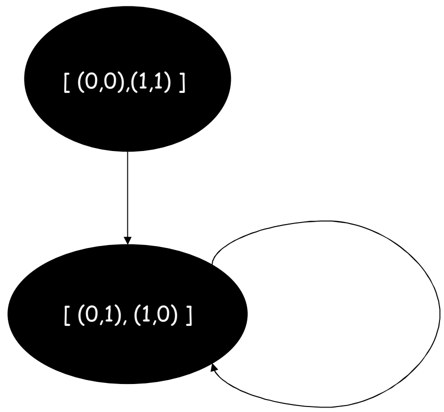
Orbits

[(0,0), (1,1)]

[(0,1), (1,0)]



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

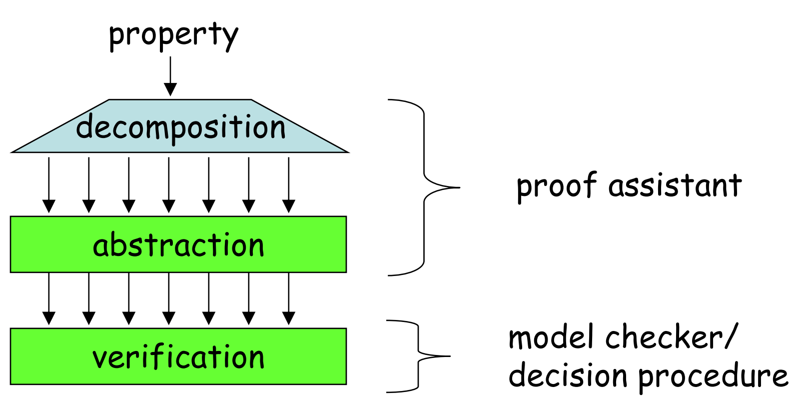
- Model checking "flat" designs/programs does not scale
  - Can be applied locally, to small modules
  - Globally to simplified models
- Model checking simplified, flat designs is mainly a "best-effort debugging" tool

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

S. A. Seshia 19

Compositional Reasoning: Divide-and-Conquer

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



S. A. Seshia 20

Notation

Proof rule specified as:

$$\frac{A_1 \;\; A_2 \;\; A_3 \;\; \ldots \;\; A_n}{C}$$

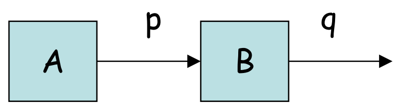
assumptions

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



$$\frac{\begin{array}{c} p \\ p \Rightarrow q \end{array}}{q}$$

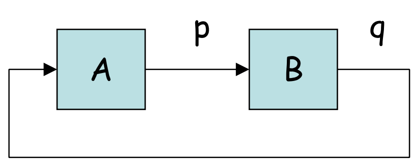
verify using A

verify using B

- 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?



$$\frac{\begin{array}{c} q \Rightarrow p \\ p \Rightarrow q \end{array}}{p \wedge q}$$

- 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

  $\neg(p \; U \; \neg q)$
- Now we can reason as follows:

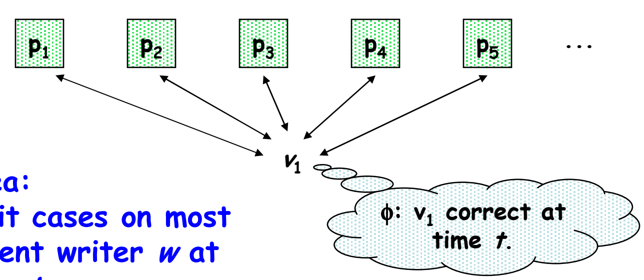
$$\frac{\begin{array}{c} q \rightarrow p \\ p \rightarrow q \end{array}}{Gp \wedge Gq}$$

verify using A

verify using B

That is, A only has to "behave" as long as B does, and vice-versa.

Temporal case splitting

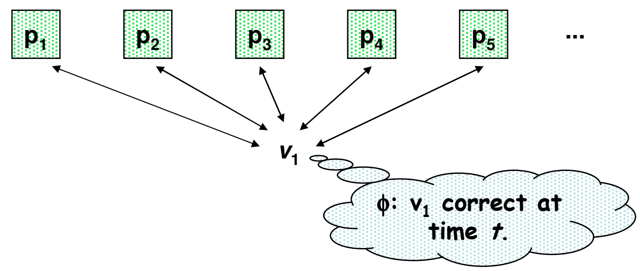


Idea:
Split cases on most recent writer *w* at time *t*.

$$\frac{\phi \; \wedge \forall i{:}\; G((w{=}i) \Rightarrow \phi)}{G\phi}$$

Rule can be used to focus within large process arrays
… but still need to deal with interdependencies

Combine with circular reasoning

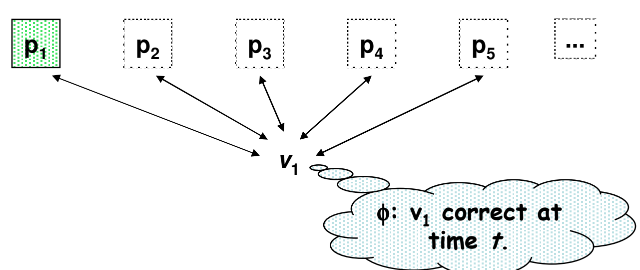


To prove case $w = i$ at time t, assume general case up to t-1:

$$\frac{\phi \wedge \forall i\colon G(\phi \Rightarrow ((w=i) \Rightarrow X\phi)}{G\phi}$$

still have many cases to prove…

Reduction by symmetry



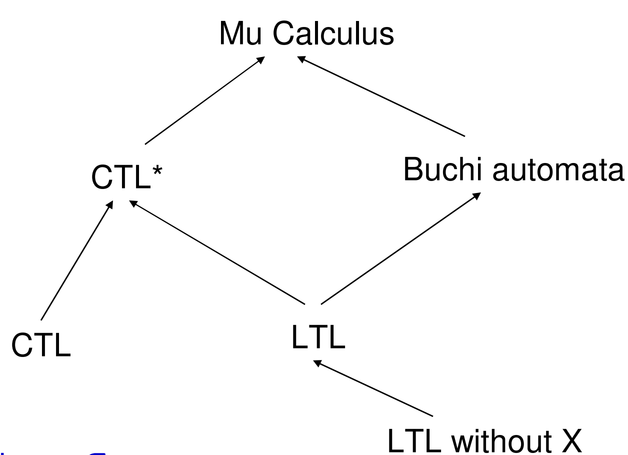
By symmetry, suffices to prove that writes by p_1 are O.K.:

$$\frac{\phi \wedge G(\phi \Rightarrow ((w=1) \Rightarrow X\phi)}{G\phi}$$

verify using p_1

The Mu-Calculus

Property Hierarchy



Legend: $\subseteq$

The Mu-Calculus

A recursive language for writing symbolic model-checking algorithms

$$EF\ a = \mu Z\, (a \vee EX\ Z)$$

$$AG\ a = \nu Z\, (a \wedge AX\ Z)$$

S. A. Seshia 31

Mu-Calculus Syntax

$$
\begin{aligned}
\varphi \quad ::= \quad & a \;|\; \neg a \;|\; Z \;| \\
& \varphi \wedge \psi \;|\; \varphi \vee \psi \;| \\
& EX\ \varphi \;|\; AX\ \varphi \;| \\
& \mu Z\ \varphi \;|\; \nu Z\ \varphi \;|
\end{aligned}
$$

Z : region variable

Any predicate transformer thus expressed is monotonic, hence all fixed points exist

S. A. Seshia 32

Mu-Calculus Semantics

$[[\ a\ ]]_{Env}$:= <a>

$[[\ \neg a\ ]]_{Env}$:= $\Sigma \setminus$ <a>

$[[\ \varphi \wedge \psi\ ]]_{Env}$:= $[[\ \varphi\ ]]_{Env} \cap [[\ \psi\ ]]_{Env}$

$[[\ \varphi \vee \psi\ ]]_{Env}$:= $[[\ \varphi\ ]]_{Env} \cup [[\ \psi\ ]]_{Env}$

$[[\ EX\ \varphi\ ]]_{Env}$:= pre($[[\ \varphi\ ]]_{Env}$)

$[[\ AX\ \varphi\ ]]_{Env}$:= $\forall$pre($[[\ \varphi\ ]]_{Env}$)

Env maps each region variable to a region

Σ is the universe

pre and $\forall$pre compute set of previous states

Operational Semantics of Mu-Calculus

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

$[[\ \nu Z\ \varphi\ ]]_E$:= S' := Σ;
repeat S := S'; S' := $[[\varphi]]_{E(Z \to S)}$ until S'=S;
return S

Model checking works as above

Complexity

- Every μ/ν alternation adds expressiveness
- Buchi automata in alternation depth of 2
- Model checking complexity:
  $O((|\varphi| \cdot N)^d)$

  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