Formal Methods in Software Development

O

Ivano Salvo and Igor Melatti

Computer Science Department



SAPIENZA UNIVERSITÀ DI ROMA

Lesson **10**, November 26th, 2019

Lesson 10a:

Data Abstraction

Data Structures

Probably, the most famous `abstraction mechanism' in checking correctness is the `**casting out nine**' method to verify multiplication: it is based on properties of congruence modulo 9, but remember... when it works, the multiplication **is not guaranteed to be correct!** You are just sure that the multiplication is wrong when it doesn't work. ©

When systems contain data structures, they often becomes "infinite" state or huge.

Abstraction is achieved by means of a map from a concrete domain *D* to an abstract domain *A*. This induces an abstraction notion among systems (*e.g.*, Kripke structures).

As usual, the goal is to generate directly abstract systems, possibly combining abstraction process with compilation.

Example of abstraction - I

Let us suppose that a variable *x* ranges over the domain D_x of integers and that we are interested in expressing properties involving the sign of *x*. We can consider the abstract domain $A_x = \{a_0, a_+, a_-\}$ and the mapping

$$h_x(d) = \begin{cases} a_0 & \text{if } d = 0\\ a_+ & \text{if } d > 0\\ a_- & \text{if } d < 0 \end{cases}$$

The abstract value of *x* can be expressed using just 3 atomic propositions $\underline{x} = a_0$, $\underline{x} = a_+$, and $\underline{x} = a_-$ where \underline{x} is an abstract value/variable.

We can no longer express properties on the exact value of x, but if abstract values are enough for the problem at hand, then we obtain a considerable state space reduction.

Abstract values induce a new Kripke structure (also the set of initial states and the transition relation are abstracted).

Reduced Kripke structures

Definition: Let $\mathcal{M}=(S, I, R, L)$ and suppose, w.l.o.g. that $S=D^n$, for some domain D. Let $h : D \rightarrow A$ the abstraction function and consider the set of atomic propositions $\underline{x}_i = a$ for some $a \in A$. We define the reduced $\mathcal{M}_R = (S_R, I_R, R_R, L_R)$ as follows:

- $S_R = \{ L(s) \mid s \in S \}, i.e., \text{ te set of all labeling}$
- $I_R = \{ L(s) \mid s \in I \}$
- *L_R*(*s*) = *s*, since states themselves are the set of atomic propostions they satisfy
- $R_R(\underline{s}, \underline{t})$ iff $\exists s, t \in S, R(s, t)$ and $\underline{s} = L(s)$ and $\underline{t} = L(t)$

 \mathcal{M}_{R} is the abstract version of \mathcal{M} and it is completely determined by the choice of abstract values and the mapping function *h* from *D* to *A*.

It easy to see that $H = \{ (s, \underline{s}) \mid s \in S \}$ is a simulation relation. ACTL* properties valid for \mathcal{M} are also valid for \mathcal{M}_R .

Example of abstraction - II

Let us consider a simple traffic light and the abstraction

- *D* = {yellow, red, green}
- $A = \{go, stop\}$

h(yellow)=h(red)=stop andh(green)=go.

Observe that in the abstraction there is **some spurious behavior**: in this case the loop in state stop: there is no loop red \rightarrow yellow \rightarrow red \rightarrow yellow \rightarrow red \rightarrow ... in the original structure.



Original Structure



Approximations - I

In many cases, \mathcal{M}_R is still too large to fit in memory and/or to be checked in reasonable time. A further idea is to build an approximation $\mathcal{M}_R \leq \mathcal{M}_A$ hopefully sufficiently close to \mathcal{M}_R to verify interesting properties.

We are given a Kripke structure $\mathcal{M} = (S, I, R, L)$ with $S = D^n$, we are given a surjective abstraction function $h : D \rightarrow A$, I and R are first order formula over variables $x_1, ..., x_n$ ranging over D.

Let $s=(d_1, ..., d_n)$, that is in state s, x_i has value d_i . Let $a_i = h(d_i)$. An atomic proposition of the form $\underline{x}_i = a_i$ denotes that the variable x_i has value a_i . $L(s) = \{\underline{x}_1 = a_1, ..., \underline{x}_n = a_n\}$.

We define \mathcal{M}_R over the abstract states $A \times ... \times A$ over variables $\underline{x}_1, ..., \underline{x}_n, \underline{x'}_1, ..., \underline{x'}_n$.

$$\underline{I} = \exists x_1, \dots, x_n \ (h(x_1) = \underline{x}_1 \land \dots \land h(x_n) = \underline{x}_n \land I(x_1, \dots, x_n))$$

$$\underline{R} = \exists x_1, \dots, x_n, x'_1, \dots, x'_n \ (h(x_1) = \underline{x}_1 \land \dots \land h(x_n) = \underline{x}_n \land h(x'_1) = \underline{x'}_1 \land \dots \land h(x'_n) = \underline{x'}_n \land R(x_1, \dots, x_n, x'_1, \dots, x'_n))$$

Approximations- II

The notation [•] is a shorthand for the existential abstraction:

$$[\phi](\underline{x}_1, \ldots, \underline{x}_n) = \exists x_1, \ldots, x_n \land h(x_i) = \underline{x}_i \land \phi(x_1, \ldots, x_n))$$

With this notation, $\underline{R} = [R]$ and $\underline{I} = [I]$.

Apply the transformation [*R*] and [*I*] may be **computationally expensive**. It is better to apply to simplified formulas. We define:

 $\mathcal{A}(P(x_1, \dots, x_m)) = [P](\underline{x}_1, \dots, \underline{x}_m) \ \mathcal{A}(\neg P(x_1, \dots, x_m)) = [\neg P](\underline{x}_1, \dots, \underline{x}_m)$ $\mathcal{A}(\phi_1 \land \phi_2) = \mathcal{A}(\phi_1) \land \mathcal{A}(\phi_2) \qquad \mathcal{A}(\phi_1 \lor \phi_2) = \mathcal{A}(\phi_1) \lor \mathcal{A}(\phi_2)$ $\mathcal{A}(\exists x \ \phi) = \exists \underline{x} \mathcal{A}(\phi) \qquad \mathcal{A}(\forall x \ \phi) = \forall \underline{x} \mathcal{A}(\phi)$

 \mathcal{A} pushes existential quantification inward, so that [\cdot] is applied to the innermost level.

However $[\phi]$ implies $\mathcal{A}(\phi)$ but it is not equivalent.

Approximations - III

Theorem: $[\phi] \Rightarrow \mathcal{A}(\phi)$. In particular $[I] \Rightarrow \mathcal{A}(I)$ and $[R] \Rightarrow \mathcal{A}(R)$. **Proof**: Induction on ϕ .

If $\phi \equiv P(x_1, ..., x_m)$, $[\phi] = \mathcal{A}(\phi)$ and the statement holds.

If $\phi \equiv \phi_1 \land \phi_2$ then $[\phi_1 \land \phi_2]$ is identical to the formula $\exists x_1, ..., x_n(\land_i h(x_1) = \underline{x}_1 \land \phi_1 \land \phi_2)$. This formula implies (but it is not equivalent to) $\exists x_1, ..., x_n(\land_i h(x_1) = \underline{x}_1 \land \phi_1) \land \exists x_1, ..., x_n(\land_i h(x_1) = \underline{x}_1 \land \phi_2)$.

If $\phi \equiv \forall x \phi'$ then $[\forall x \phi']$ is (we can assume $\forall i$ variable $x \neq x_i$) $\exists x_1, ..., x_n (\land_i h(x_i) = \underline{x}_i \land \forall x \phi'(x, x_1, ..., x_n))$ $(x \neq x_i) \equiv \exists x_1, ..., x_n \forall x (\land_i h(x_i) = \underline{x}_i \land \phi'(x, x_1, ..., x_n))$ (this is not eq.) $\Rightarrow \forall x \exists x_1, ..., x_n (\land_i h(x_i) = \underline{x}_i \land \phi'(x, x_1, ..., x_n))$ $(h \operatorname{srj}) \equiv \forall \underline{x} \exists x [\exists x_1, ..., x_n (h(x) = \underline{x} \land_i h(x_i) = \underline{x}_i \land \phi'(x, x_1, ..., x_n))]$ $\equiv \forall \underline{x} [\phi']$. By IND $[\phi'] \Rightarrow \mathcal{A}(\phi')$ and hence $\forall \underline{x} [\phi'] \Rightarrow \forall \underline{x} \mathcal{A}(\phi')$

 $\phi \equiv \phi_1 \lor \phi_2$ and $\phi \equiv \exists x \phi'$ are similar.

Approximations -IV

Theorem: $\mathcal{M} \leq \mathcal{M}_A$

Proof: Let $s = (d_1, ..., d_n)$ and $s_a = (a_1, ..., a_n)$. We define $H(s, s_a)$ iff for all *i* we have $a_n = h(d_i)$.

First of all, *s* and *s*_{*a*} have the same labelling.

Assume R(s, t) with $t = (e_1, ..., e_n)$. Define $t_a = (h(e_1), ..., h(a_n))$. We have to show that $R_A(s_a, t_a)$. (*s*, *t*) correspond to valuation satisfying *R*. We show that $[R](s_a, t_a)$. By def of $[\cdot], [R](s_a, t_a)$ holds iff:

$$\exists x_1, \ldots, x_n, x'_1, \ldots, x'_n (h(x_1) = h(d_1) \land \ldots \land h(x_n) = h(d_n) \land \land h(x'_1) = h(e_1) \land \ldots \land h(x'_n) = h(e_n) \land R(x_1, \ldots, x_n, x'_1, \ldots, x'_n))$$

R(s, t) holds by taking d_i as witness for x_i and e_i as witness for x'_i and hence $[R](s_a, t_a)$. By the previous theorem, this implies that $\mathcal{A}(R)(s_a, t_a)$ is true and $\mathcal{A}(R)$ defines \mathcal{M}_A . Thus H is a simulation between \mathcal{M} and \mathcal{M}_A . Similar for initial states.

Exact Approximations

 $\mathcal{M} \leq \mathcal{M}_A$ implies that every ACTL* formula satisfied by \mathcal{M}_A also holds in \mathcal{M} . Here we sketch properties ensuring that $\mathcal{M} \cong \mathcal{M}_A$.

An abstraction $h : D \rightarrow A$ induces an equivalence on D defined by $d \sim d'$ iff h(d) = h(d').

The equivalence ~ is a congruence w.r.t. a primitive relation *P* **iff** $\forall d_1, ..., d_n, e_1, ..., e_n \land_i d_i \sim e_i \rightarrow P(d_1, ..., d_m) \Leftrightarrow P(e_1, ..., e_m)$

Theorem: If ~ is a congruence wrt to primitive relations in ϕ , then $[\phi] \Leftrightarrow \mathcal{A}(\phi)$. In particular, $[R] \Leftrightarrow \mathcal{A}(R)$ and $[I] \Leftrightarrow \mathcal{A}(I)$.

Theorem: If ~ is a congruence wrt to primitive relations in \mathcal{M} , then $\mathcal{M} \cong \mathcal{M}_A$.

Lesson 10b:

Examples of Data Abstractions

Congruence modulo m

This is a common useful abstraction for systems involving arithmetic. $h(i) = i \mod m$. This abstraction is a congruence:

 $(i \mod m) + (j \mod m) = i + j \pmod{m}$ $(i \mod m) - (j \mod m) = i - j \pmod{m}$ $(i \mod m)(j \mod m) = i j \pmod{m}$

The value modulo *m* depends on values modulo *m* of operands.

Theorem [Chinese Remainder Theorem]: Let $m_1, m_2, ..., m_n$ be pairwise relatively prime positive integers. Let $m = m_1m_2...m_n$ and let $b, i_1, i_2, ..., i_n$ be integers. Then there is a unique i such that: $b \le i < b+m$ and $i \equiv i_j \pmod{m_j}$ for all $1 \le j < n$.

This theorem essentially says that we can infer the value of *i* by considering equivalence classes of $(\mod m_j)$ for all $1 \le j < n$.

Verification of a multiplier

Verifying a 16-bit multiplier could be impractical (see code next slide). The program (circuit) has 3 input: *req* (that is a request signal starting the execution), *in1* and *in2* (factors to operate on)

The multiplier performs a sequence of shift and add steps until *factor1* is 0 or an *overflow* has been generated.

The specifications is a series of formulas (see next slides) that checks congruence modulo *m* of the product result, according to abstraction.

Observe that the specification admits the possibility that the multiplier outputs always an overflow.

$AG(waiting \wedge req \wedge (in1 \mod m = i) \wedge (in2 \mod m = j)$

dimension

input in1 : 16; input in2 : 16; input req : 1; output factor1 : 16 := 0; output factor2 : 16 := 0; output output : 16 := 0; output overflow : 1 := 0; output ack : 1 := 0;

procedure waitfor(e) while ¬e wait; end while; end procedure;

loop

 $\rightarrow \mathbf{A}[\neg ack \ \mathbf{U} \ ack \land (overflow \lor (output \ \mathrm{mod} \ m = ij \ \mathrm{mod} \ m))])$

In the verification, *factor1*, *in1*, *in2*, and *output* are abstracted modulo *m*

Verification can be performed with m = 5, 7, 9, 11, 32 whose product is 110880

loop

if $(factor 1 = 0) \lor (overflow = 1)$ then break; if lsb(factor 1) = 1 then (overflow, output) := (output: 17) + factor 2; $factor 1 := factor 1 \gg 1;$ right shift wait; if $(factor 1 = 0) \lor (overflow = 1)$ then break; $(overflow, factor 2) := (factor 2: 17) \ll 1;$ wait; end loop; ack := 1;wait; synchronous execution waitfor($\neg req$); ack := 0;end loop;

Logarithm representation

When only the order of magnitude is important, it is useful to represent a quantity by its logarithm. Define: $\lg i = [\log_2(i + 1)]$, that is the smallest number of digits needed to represent i > 0.

Let us consider again the 16-bit multiplier. A circuit that always return overflow satisfies the specification we consider.

Observe that if $\lg i + \lg j \le 16$ then $\lg i j \le 16$ and the multiplication should not overflow. Conversely, if $\lg i + \lg j \ge 18$ then $\lg i j \ge 17$ and the multiplication will overflow. If $\lg i + \lg j = 17$, this test is inconclusive.

We can strengthen our specification to include the following, that can be checked abstracting all 16 bit variables with their logarithms:

 $\mathbf{AG}(\text{waiting} \land \text{req} \land (\lg \text{ in } l + \lg \text{ in } 2 \le 16) \rightarrow \mathbf{A}[\neg ack \ \mathbf{U} \ ack \land \neg overflow])$ $\mathbf{AG}(\text{waiting} \land \text{req} \land (\lg \text{ in } l + \lg \text{ in } 2 \ge 18) \rightarrow \mathbf{A}[\neg ack \ \mathbf{U} \ ack \land overflow])$

Single bit & Product Abstractions

When bitwise logical operations are involved in a system, the following abstraction may be useful: $h(i) = j^{th}$ bit of *i*.

Moreover, if h_1 and h_2 are two abstractions, then also $h(i) = (h_1(i), h_2(i))$ is an abstraction.

As in the case of multiplier, two abstractions may make possible to verify properties that are not verifiable using just one abstraction.

The program in the next slide computes the parity of a 16-bit input. It should meet the following properties (let #i to be true if the parity of i is odd):

- The value assigned to *b* has the same parity of the input *in*
- *#b* ⊕ *parity* is invariant

Single bit & Product Abstractions

The above properties can be expressed by the following CTL formula:

 \neg #in \land **AX** (\neg #b \land **AG** \neg (#b \oplus parity)) \lor #in \land **AX** (#b \land **AG** (#b \oplus parity))

This property can be verified by using a combined abstraction on variables *in* and *b*.

Values of this variables can be grouped both by the value of their low-order bit and their parity.

Using these abstractions, verification takes few seconds only. input in : 16; output parity : 1 := 0; output b : 16 := 0; output done : 1 := 0;

```
b := in;

wait;

while b \neq 0 do

parity := parity \oplus lsb(b);

b := b \gg 1;

wait;

end while;

done := 1;
```

Symbolic Abstractions

The use of OBDDs makes it possible to use abstractions that depend on symbolic values.

As a simple example, consider the program on the right: The next state of *b* is always equal to the current state of *a*. We state this property for a fix value, say 42. **input** *a* : 8; **output** *b* : 8 := 0;

To verify the property AG ($a=42 \rightarrow AX b=42$) we can use the following abstraction:

$$h(d) = \begin{cases} 0 & \text{if } d = 42\\ 1 & \text{otherwise} \end{cases}$$

The above property becomes **AG** ($a=0 \rightarrow AX b=0$) and can be easily checked using 1 digit variables. Of course, we do not want to repeat the verification for each integer value!

Symbolic Abstractions

We can consider the parametrized abstraction, that leads to a parametrized transition relation:

$$h_c(d) = \begin{cases} 0 & \text{if } d = c \\ 1 & \text{otherwise} \end{cases}$$

We can perform symbolic model checking as follows:

- 1. Use an OBDD to represent h_c (supplied by the user);
- 2. Compile with h_c to get an OBDD representing $R_c(\underline{a}, \underline{a'}, \underline{b}, \underline{b'}, c)$

3. Generate the parametrized state set: the model checker views *c* just as a state component that does not change.

4. Possibly, to generate a counterexample choose a specific *c*.

There is a slightly more complicated example on the Clarke book.

Lesson 10c:

Symmetry

Groups

We recall basic definitions about groups.

Definition: A **group** (G, \cdot) is a set with a binary operation $\cdot : G \times G \rightarrow G$, such that:

- 1. is *associative*, that is $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- 2. there is an *identity* $e \in G$, that is $a \cdot e = e \cdot a = a$
- 3. Each $a \in G$ has an *inverse* a^{-1} , such that $a \cdot a^{-1} = a^{-1} \cdot a = e$

Definition: Given a set of elements $\{g_1, ..., g_k\} \subseteq G$, we indicate with $\langle g_1, ..., g_k \rangle$ the smallest subgroup *H* of *G* containing $g_1, ..., g_k$, that are called **generators** of *H*. *H* is the minimum set closed under • and inverse.

Groups of permutations

Definition: A **permutation** σ on a finite set A is a bijection $\sigma : A \rightarrow A$. We call $dom \sigma = \{a \in A \mid \sigma(a) \neq a\}$. Two permutations σ_1 and σ_2 are **disjoint** if $dom \sigma_1 \cap dom \sigma_2 = \emptyset$. The set of all permutations on A is the **permutation group** *Sym* A, where the operation is function composition. **Identity function** is the identity and the **inverse function** is the inverse.

A permutation that maps $x = x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n = x$ is called a **cycle**, denoted by $(x_1 x_2 \dots x_n)$. A cycle of length 2 is called **transposition**.

Each permutation can be written as the composition of **disjoint cycles** or as the composition of transposition (not disjoint).

Example: Let us consider $A = \{1, 2, 3, 4, 5\}$ and σ defined by $\{(1,3), (2,4), (3,1), (4,5), (5,2)\}$. Then $\sigma = (1 3)^{\circ}(2 4 5)$ and $\sigma = (1 3)^{\circ}(2 5)^{\circ}(2 4)$. The subgroup generated by the two permutations (1 3) and (2 4 5) is the group $\{e, (1 3), (2 4 5), (1 3)^{\circ}(2 4 5), (2 5 4), (1 3)^{\circ}(2 5 4)\}$. Observe $(2 5 4)=(2 4 5)^{\circ}(2 4 5)$

Automorphism of Kripke struct.

Definition: Let $\mathcal{M} = (S, R, L)$ be a Kripke structure and let G be a permutation group on the state space $S. \sigma \in G$ is an **automorphism** iff it preserves R, that is:

 $\forall s, t \in S. R(s, t) \Rightarrow R(\sigma(s), \sigma(t)) \quad (\texttt{*})$

If for all $\sigma \in G$, σ is an automorphism, G is called an **automorphism group** of \mathcal{M} . Since automorphisms have inverse in G, (*) is equivalent to: $\forall s, t \in S$. $R(s, t) \Leftrightarrow R(\sigma(s), \sigma(t))$.

If we take a set of automorphisms as generators, the generated subgroup is an automorphism group (easy, as the composition of two atomorphisms and the inverse are automorphisms).

Observation: Since *L* does not come into play, we essentially define graph automorphisms.

Example of Kripke automorphism

Let us consider a **token ring** algorithm with a process Q and many processes P_i . Q and P_i have the same structure: three states: n (non-critical), t (has the token) and c (critical). There are two visible actions: r (receive token) and s (send token). Initially, Q is in state t and P_i are in state n. Composition must syncrhonise on visible actions r and s.



Example of Kripke automorphism

Let us consider the permutation σ on states of P | Q defined by $\sigma(n, t)=(t, n), \sigma(t, n)=(n, t), \sigma(n, c)=(c, n), and \sigma(c, n)=(n, c).$ Examining transitions, it easy to see that σ is an automorphism.

For semplicity, we can consider $S = D^n$, defined by *n* variables. For example, $Q | P^i = Q | P | ... | P$ can be represented by *i*+1 variables over the domain $D = \{n, t, c\}$ of process states. It is usually to define auotorphisms as permutations of state variable indices. For example, the above automorphism is the transposition (1 2).

A permutation σ on $\{1, 2, ..., n\}$ defines a permutation σ' on S, defined by $\sigma'(x_1, ..., x_n) = (x_{\sigma(1)}, ..., x_{\sigma(n)})$. To see that σ' is indeed a permutation, just observe that $x \neq y$ implies $\sigma'(x) \neq \sigma'(y)$.

Quotient Models

Definition: Let *G* be a permutation group on S and let $s \in S$. The **orbit** of *s* is $\theta(s) = \{t \mid \exists \sigma \in G, \sigma(s) = t\}$. From each orbit $\theta(s)$ we can choose a representative $rep(\theta(s))$.

Definition: Let $\mathcal{M} = (S, R, L)$ be a Kripke structure and let G be an automorphism group acting on S. G is an **invariance group** for an atomic proposition p iff

$$(\forall \sigma \in G) \ (\forall s \in S) \ p \in L(s) \Leftrightarrow p \in L(\sigma(s))$$

Definition: Let $\mathcal{M} = (S, R, L)$ be a Kripke structure and let G be an invariance group acting on S. The **quotient structure** $\mathcal{M}_G = (S_G, R_G, L_G)$ is defined by:

- $S_G = \{ \theta(s) \mid s \in S \}$
- $R_G = \{ (\theta(s), \theta(s')) \mid (s, s') \in R \}$
- $L_G(\theta(s)) = L(rep(\theta(s)))$

Back to the token ring example

Let us consider again the token ring example and the permutation group $G = \langle (1 \ 2) \rangle$ on the states of Q | P. Orbits induced by G are $\{(t, n), (n, t)\}$ and $\{(c, n), (n, c)\}$. The resulting quotient model in the picture.

Interestingly, if we consider $Q | P^i$, the Kripke structure has 2(i+1) reachable states. Taking $G = \langle (1 \ 2 \ 3 \ ... \ i+1) \rangle$ induces only two orbits: $\{(t, n^i), (n, t, n^{i-1}), ..., (n^i, t)\}$ and $\{(c, n^i), (n, c, n^{i-1}), ..., (n^i, c)\}$ and thus the quotient of $Q | P^i$ is identical to that of Q | P.



Properties of quotient models

Lemma: Let $\mathcal{M} = (S, R, L)$ be a Kripke structure over AP. Let G be an invariance group for all $p \in AP$ and let \mathcal{M}_G be the quotient model. Then, the relation $B = \{(s, \theta(s)) \mid s \in S\}$ is a bisimulation between \mathcal{M} and \mathcal{M}_G on and let $s \in S$.

Proof: Since *G* is an invariant group for *AP*, $L(s)=L(\theta(s))$.

R(s, t) implies $R_G(\theta(s), \theta(t))$ (def. of R_G) and $B(t, \theta(t))$ (def. of B).

Finally, $R_G(\theta(s), \vartheta)$: let $t = rep \, \vartheta, \, \vartheta = \theta(t)$ and hence $R_G(\theta(s), \vartheta)$ can be rewritten as $R_G(\theta(s), \theta(t))$. This implies that there exist two states s', t' such that R(s', t') and $s' \in \theta(s)$ and $t' \in \theta(t)$. Since s and s' (and t and t') belong to the same orbit, there exist $\sigma_1, \sigma_2 \in G$ such that $\sigma_1(s') = s$ and $\sigma_2(t') = t$, that in turn implies $R(\sigma_1(s'), \sigma_2(t'))$. \Box

Corollary: Let \mathcal{M} be a Kripke structure over AP. Let G be an invariance group for AP. Then for every $s \in G$ and every CTL^* formula f, we have: $\mathcal{M}, s \models f \Leftrightarrow \mathcal{M}_G, \theta(s) \models f$.

Model Checking with Simmetry

In the presence of symmetry only representative are considered. Here, we present an explicit algorithm for reachability, that assume the existence of a function $\xi(q)$ that associate to each state q, the unique representative of q.

This simple reachability algorithm can be extended to full CTL model checking.

With OBDDs things are a bit more complex.

reached := \emptyset ; unexplored $:= \emptyset;$ for all initial states s do **append** $\xi(s)$ to *reach*; append $\xi(s)$ to unexplored; end for all while unexplored $\neq \emptyset$ do **remove** a state *s* from *unexplored*; for all successor states q of s do if $\xi(q)$ is not in reached **append** $\xi(q)$ to reached; **append** $\xi(q)$ to *unexplored*; end if end for all end while

Model Checking with Simmetry

If we have an OBDD for $R(v_1, ..., v_k, v_1', ..., v_k')$, and a permutation σ it is easy to check that σ is an automorphism, just checking if $R(v_1, ..., v_k, v_1', ..., v_k')$ and $R(v_{\sigma(1)}, ..., v_{\sigma(k)}, v'_{\sigma(1)}, ..., v'_{\sigma(k)})$ are **identical**.

Having generators $g_1, ..., g_r$, the orbit relation $\Theta(x, y) = x \in \theta(y)$ can be computed as the minimum fixpoint of the equation:

$$Y(x, y) = (x = y \lor \exists z (Y(x, z) \land \lor_i y = g_i(z)))$$

Having Θ , $\xi : S \to S$ can be computed, for example, by choosing the state whose sequence of bit in its representation is the smallest in the lexicographic order.

Having ξ the transition relation R_G can be computed as:

$$R_G(x, y) = \exists w \ z \ (\mathbb{R}(w, z) \land x = \xi(w) \land y = \xi(z))$$

Lesson 10

That's all Folks...

... Questions?