Formal Methods in Software Development

Counteracting State Explosion Problem III: Bisimulation

Ivano Salvo

Computer Science Department

SAPIENZA UNIVERSITÀ DI ROMA

Lesson **10**, December 7^{th}, 2020

Lesson 10a:

Data Abstraction

Data Structures

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:

- when it works, the multiplication **is not guaranteed to be correct!**
- You are 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).

Goal: **generate directly abstract systems**, possibly combining abstraction process with compilation.

Example of abstraction - I

Let x be a variable ranging over integers and that the property we are interested in involves just 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 is expressed **by using just 3 atomic propositions**: '$\underline{x}= a_0$', '$\underline{x}= a_+$', and '$\underline{x}= a_-$' where $\underline{x}$ is the 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**.

Spurios behaviour can be introduced: the sum of a negative and positive could be either positive or negative.

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 system** $\mathcal{M}_R$=(S_R, I_R, R_R, L_R) as follows:

- $S_R = \{ L(s) \mid s \in S \}$, *i.e.*, 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 $\underline{t} = L(t)$

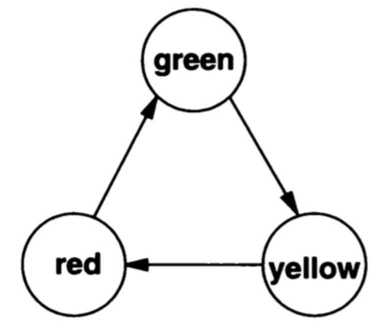
$\mathcal{M}_R$ is the abstract version of $\mathcal{M}$ and it **is completely determined by the choice of abstract values** and the mapping function h from D to A.

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

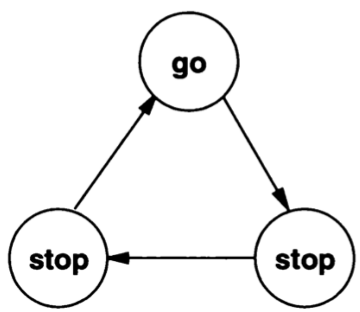
Example of abstraction - II

A simple traffic light with values D = {yellow, red, green}

and the abstract domain A = {go, stop}
with the abstraction function h defined by:
h(yellow)=h(red)=stop and h(green)=go.

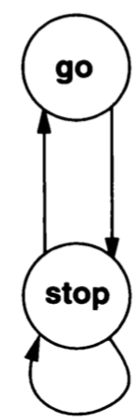


Original Structure



Structure M

The reduced model is:



Structure M_r

Observe that in the abstraction there is a **spurious behavior**: the loop in state stop.
There is no loop red→yellow→red →… →yellow→red→ … in the original structure.

Approximations - I

$\mathcal{M}_R$ can be **still too large** to fit in memory and/or to be checked in reasonable time. A further **idea** is to build **an approximation** $\mathcal{M}_R \preccurlyeq \mathcal{M}_A$ close enough to $\mathcal{M}_R$ to verify interesting properties.

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

Let $s = (d_1, \ldots, d_n)$, that is in s, each 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 (abstract) value a_i. $L(s)$={\`$\underline{x}_1$=$a_1$\`, …, \`$\underline{x}_n$=$a_n$\`}.

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

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

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

free variables of $\underline{I}$ and $\underline{R}$ are abstract variables!

Approximations- II

We use the notation [·] as a shorthand for the existential abstraction:

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

For example, $\underline{R} = [R]$ and $\underline{I} = [I]$.

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

$\mathcal{A}(P(x_1,\ldots, x_m)) = [P](\underline{x}_1, \ldots, \underline{x}_m)$ $\mathcal{A}(\neg P(x_1, \ldots, x_m)) = [\neg P](\underline{x}_1,\ldots, \underline{x}_m)$

$\mathcal{A}(\phi_1 \wedge \phi_2) = \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$ $\mathcal{A}(\phi_1 \vee \phi_2) = \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$

$\mathcal{A}(\exists x\, \phi) = \exists \underline{x} \mathcal{A}(\phi)$ $\mathcal{A}(\forall x\, \phi) = \forall \underline{x} \mathcal{A}(\phi)$

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

Be careful: $[\phi]$ 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 $\boldsymbol{\phi \equiv P(x_1, \ldots, x_m)}$, $[\phi]=\mathcal{A}(\phi)$ and the statement holds.
- If $\boldsymbol{\phi \equiv \phi_1 \wedge \phi_2}$ then $[\phi_1 \wedge \phi_2]$ is identical to the formula $\exists x_1, \ldots, x_n(\bigwedge_i h(x_1)= \underline{x}_1 \wedge \phi_1 \wedge \phi_2)$.

This formula implies (**but not equivalent to**) $\exists x_1, \ldots, x_n(\bigwedge_i h(x_1)= \underline{x}_1 \wedge \phi_1) \wedge \exists x_1, \ldots, x_n(\bigwedge_i h(x_1)= \underline{x}_1 \wedge \phi_2)$ [**it is easier to find two witness for $\exists$**].

Moreover, by (**IND**) we have that $[\phi_1] \Rightarrow \mathcal{A}(\phi_1)$ and $[\phi_2] \Rightarrow \mathcal{A}(\phi_2)$ and. Therefore $[\phi_1 \wedge \phi_2]$ implies $\mathcal{A}(\phi_1 \wedge \phi_2)$.

- If $\boldsymbol{\phi \equiv \forall x\, \phi'}$ then $[\forall x\, \phi']$ is (we can assume $\forall i.\ x \neq x_i$)

$\exists x_1, \ldots, x_n(\bigwedge_i h(x_i)= \underline{x}_i \wedge \forall x\, \phi'(x, x_1, \ldots, x_n)) \equiv$

$(x \neq x_i) \equiv \exists x_1, \ldots, x_n \forall x\, (\bigwedge_i h(x_i)= \underline{x}_i \wedge \phi'(x, x_1, \ldots, x_n)) \Rightarrow$

$\Rightarrow$ ***(this is not eq.)*** $\forall x \exists x_1, \ldots, x_n\, (\bigwedge_i h(x_i)= \underline{x}_i \wedge \phi'(x, x_1, \ldots, x_n)) \equiv$

$(h \text{ srj}) \equiv \forall \underline{x} \exists x\, [\exists x_1, \ldots, x_n(h(x)= \underline{x} \bigwedge_i h(x_i)= \underline{x}_i \wedge \phi'(x, x_1, \ldots, x_n))]$

$\equiv \forall \underline{x}\, [\phi']$. By **IND** $[\phi'] \Rightarrow \mathcal{A}(\phi')$ and hence $\forall \underline{x}\, [\phi'] \Rightarrow \forall \underline{x} \mathcal{A}(\phi')$

$\phi \equiv \phi_1 \vee \phi_2$ and $\phi \equiv \exists x\, \phi'$ are similar. $\square$

Approximations -IV

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

Proof: Let $s = (d_1, \ldots, d_n)$ and $s_a = (a_1, \ldots, a_n)$. We define $H(s, s_a)$ iff for all i we have $a_n = h(d_i)$. In this way, s and s_a have the same labelling.

Assume $R(s, t)$ with $t = (e_1, \ldots, e_n)$. Define $t_a = (h(e_1), \ldots, h(a_n))$. We have to show that $R_A(s_a, t_a)$. The transition (s, t) corresponds to a valuation satisfying R. We show that $[R](s_a, t_a)$.

By def of $[\ \cdot\ ]$, $[R](s_a, t_a)$ holds iff:

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

$R(s, t)$ 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. $\square$

Exact Approximations

$\mathcal{M} \leqslant \mathcal{M}_A$ 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 $h(d) = h(d')$.

Definition: An equivalence ~ is a congruence w.r.t. a primitive relation P **iff** $\forall d_1, \ldots, d_n, e_1, \ldots, e_n. \bigwedge_i d_i \Rightarrow e_i \rightarrow P(d_1, \ldots, d_m) \Leftrightarrow P(e_1, \ldots, e_m)$

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 "useful" Data Abstractions

A simple language

We present some examples, modeled in a simple language for **synchronous digital circuits** as **finite state** Moore automata.

The compiler has some built-in abstraction and build an OBDD representation of the defined system.

The compiler generates directly the abstract system.

We will write abstraction by using some syntactic sugar: for example "even(x) instead of '$\underline{x} = a_{\text{even}}$'

A simple language

input/output variables
length in number of bits

```
input set : 1;
input start : 8;
output count : 8 = 0;
input alarm : 1 = 1;

loop
    if set == 1
        then count = start;
        else if count > 0 then count = count -1;
    if count == 0 then alarm = 1;
                  else alarm = 0;
    wait;
end loop;
```

Settable Countdown Timer

synchronization: outputs become visible, get inputs and a new step can take place

Congruence modulo m

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

$$(i \bmod m) + (j \bmod m) = i + j \pmod m$$

$$(i \bmod m) - (j \bmod m) = i - j \pmod m$$

$$(i \bmod m)(j \bmod m) = i\, j \pmod m$$

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

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

This theorem essentially says that **we can infer the value of *i* by considering equivalence classes of** $\pmod{m_j}$ for all $1 \leq j < n$, provided that i is known to belong to an interval of width m.

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 be multiplied).

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 **that checks congruence modulo *m* of the product result**, according to the abstraction:

$$\mathbf{AG}\ (\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j) \\ \rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge (\mathit{overflow} \vee \mathit{output} \bmod m = i\, j \bmod m)$$

```
loop
    waitfor(req)
    factor1 = in1;
    factor2 = in2;
    output = 0;
    overflow = 0;
    wait;
    loop #main multiplication loop
        if (factor1==0 ∨ overflow == 1) then break;
        if (lsb(factor1)== 1) then
            (overflow, output)=(output: 17)+factor2;
        factor1 = factor1 » 1; # right shift
        wait;
        if (factor1==0 ∨ overflow == 1) then break;
        (overflow, factor2)=(factor2: 17) « 1; # left shift
        wait; # synchronization
    ack=1;
    wait;
    waitfor(req);
    ack = 0;
```

```
# variable declarations
input in1: 16;
input in2: 16;
input req: 1;
output factor1 : 16;
output factor2 : 16;
output output : 16;
output overflow : 1;
output ack : 1;
```

```
def waitfor(e)
    while ¬e
        wait;
```

Verification of a multiplier

In the verification, *factor1*, *in1*, *in2*, and *output* **are abstracted modulo** ***m*****.** We can perform verification with $m = 5, 7, 11, 32$ whose product is 110880 that is enough for a 16-bit multiplier.

$$\mathbf{AG}\,(\mathit{waiting} \wedge \mathit{req} \wedge (\mathit{in1} \bmod m = i) \wedge (\mathit{in2} \bmod m = j)$$
$$\rightarrow \mathbf{A}(\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge (\mathit{overflow} \vee \mathit{output} \bmod m = i\,j \bmod m)$$

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

Correctness of the overflow bit **can be checked separetely using a different abstraction**.

Logarithmic representation

When only the order of magnitude is important, it is useful to represent a quantity by its logarithm. Define: lg i = $\lceil \log_2(i+1) \rceil$, 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 considered**.

Observe that if lg i + lg $j \leq 16$ then lg $i\,j \leq 16$ and the multiplication should not overflow. Conversely, if lg i + lg $j \geq 18$ then lg $i\,j \geq 17$ and the multiplication will overflow.

If lg i + lg j = 17, this test is inconclusive.

Specification can be strenghten checking overflow by abstracting **all 16 bit variables with their logarithms**:

$$\mathbf{AG}\ (wtg \wedge req \wedge (\lg in1 + \lg in2 \leq 16) \rightarrow \mathbf{A}(\neg ack\ \mathbf{U}\ ack \wedge \neg\ ovflw)$$

$$\mathbf{AG}\ (wtg \wedge req \wedge (\lg in1 + \lg in2 \geq 18) \rightarrow \mathbf{A}(\neg ack\ \mathbf{U}\ ack \wedge ovflw)$$

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 can 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*
- $\#b \oplus parity$ is invariant

Single bit & Product Abstractions

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

$\neg\#in \wedge \mathbf{AX}\,(\neg\#b \wedge \mathbf{AG}\neg(\#b \oplus parity))$
$\vee\ \#in \wedge \mathbf{AX}\,(\#b \wedge \mathbf{AG}\,(\#b \oplus parity))$

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

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

Using these abstractions, verification takes few seconds only.

```
# variable declaration
input in: 16;
output parity : 1 = 0;
output b : 16 = 0;
output done : 1 = 0;
b = in;
wait;
while b ≠ 0 do
    parity = parity ⊕ lsb(b);
    b = b » 1;
    wait;
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;

loop
    b = a;
    wait;
```

To verifiy the property **AG** (a=42 → **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 → **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

Symmetry

Systems exhibit considerable symmetry: circuits, bus protocols, and in general systems with **replicated structures** as sub-systems.

The existence of symmetries implies the existence of **non-trivial permutation groups** that can be used to define equivalence relations that preserve both state labeling and transitions.

The quotient model is **bisimilar** to the original model (equivalent with respect to checking a CTL* property) and can be **significantly smaller**.

Groups

We recall basic definitions about groups.

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

1. $\cdot$ 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 $\{g_1, \ldots, g_k\} \subseteq G$, we indicate with $\langle g_1, \ldots, g_k \rangle$ the smallest subgroup H of G containing $g_1, \ldots, g_k$, that are called **generators** of H.

H is the minimum set closed under $\cdot$ 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 $dom\ \sigma_1 \cap dom\ \sigma_2 = \varnothing$.

The set of all permutations on a set 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 $x = x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n = x$ is called a **cycle**, denoted by $(x_1\ x_2\ \ldots\ x_n)$.

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 of permutations

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 that $(2\ 5\ 4)=(2\ 4\ 5)\circ(2\ 4\ 5)$

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:

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

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: $\forall s, t \in S.\ R(s, t) \Leftrightarrow R(\sigma(s), \sigma(t))$.

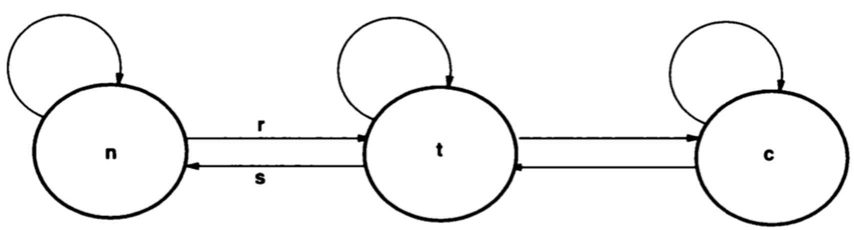
If we take a set of automorphisms as generators, the generated subgroup is an automorphism group (easy, as the composition of two atomorphisms and the inverse are automorphisms).

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

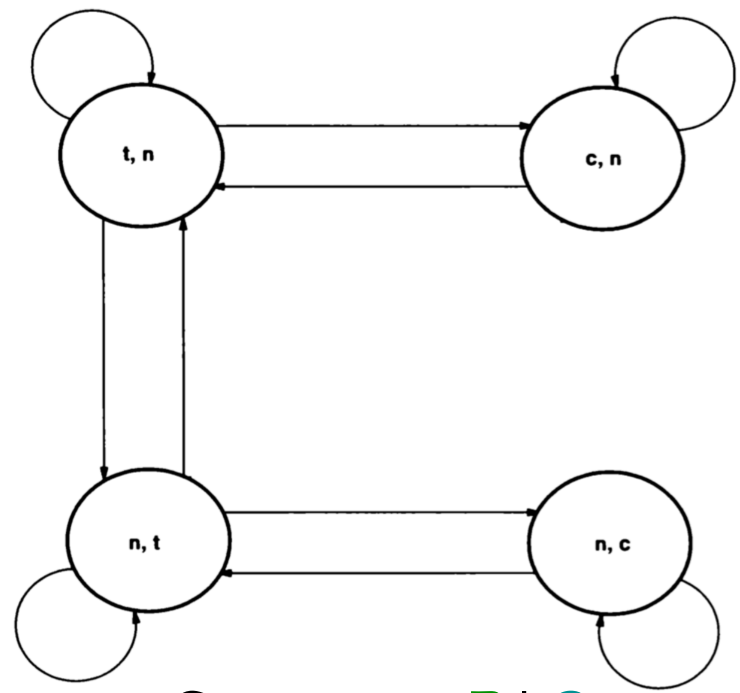
Example of Kripke automorphism

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.



Processes P_i and Q



Composition P | Q

Example of Kripke automorphism

Let us consider the permutation σ on states of $P\,|\,Q$ defined by σ(n, t)=(t, n), σ(t, n)=(n, t), σ(n, c)=(c, n), and σ(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, $Q\,|\,P^i = Q\,|\,P\,|\,\ldots\,|\,P$ 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 $\{1, 2, \ldots, n\}$ defines a permutation σ' on S, defined by $\sigma'(x_1, \ldots, x_n) = (x_{\sigma(1)}, \ldots, x_{\sigma(n)})$. 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

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

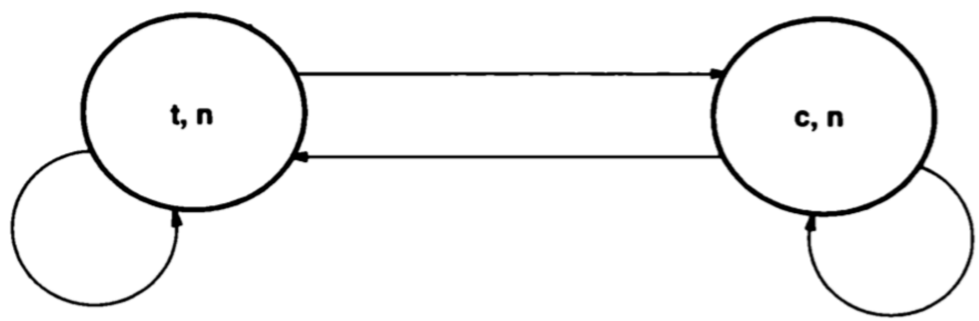
Definition: Let $\mathcal{M} = (S, R, L)$ be a Kripke structure and let G be an invariance group acting on S. The **quotient structure** $\mathcal{M}_G = (S_G, R_G, L_G)$ is defined by:

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

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



Quotient model of both $Q \mid P$ and $Q \mid P^i$

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 = { $(s, \theta(s))$ | $s \in S$ } is a bisimulation** between $\mathcal{M}$ and $\mathcal{M}_G$.

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 $R(\sigma_1(s'), \sigma_2(t'))$. $\square$

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 representatitive of q.

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

With OBDDs things are a bit more complex.

reached := ∅;
unexplored := ∅;
for all initial states *s* **do**
    append $\xi(s)$ to *reach*;
    append $\xi(s)$ to *unexplored*;
end for all
while *unexplored* ≠ ∅ **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 $R(v_1, \ldots, v_k, v_1', \ldots, v_k')$, and a permutation σ it is easy to check that σ is an automorphism, just checking if $R(v_1, \ldots, v_k, v_1', \ldots, v_k')$ and $R(v_{\sigma(1)}, \ldots, v_{\sigma(k)}, v'_{\sigma(1)}, \ldots, v'_{\sigma(k)})$ are **identical**.

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

$$\mathrm{Y}(x, y) = (x = y \vee \exists z\, (\mathrm{Y}(x, z) \wedge \bigvee_i y = g_i(z)))$$

Having Θ, $\xi : S \rightarrow S$ 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:

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

That's all Folks!

Thanks for your attention…

…Questions?