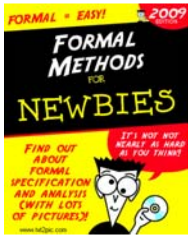
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 8: Model Checking, Part 2

Juergen Dingel
Feb, 2009

Readings:
- Spin book: Chapter 8, pages 178-181 (Search Algorithms)
- Course notes on CTL

Outline

- How to check for
  - assertion violations & deadlock
    - Basic DFS
  - safety properties
    - expressed as FSAs (in Bogor)
    - expressed as Never Claims (in Spin)
    - expressed as LTL properties (in Spin)
  - liveness properties
    - progress labels (in Spin)
    - expressed as Never Claims (in Spin)
    - expressed as LTL properties (in Spin)

Preliminaries

- Let system S be given by n concurrent threads
  $T_1, \ldots, T_n$
- Threads T_i execute asynchronously in S
- So, A_S, the FSA representing S, is obtained by building the asynchronous composition of the A_{Ti}, the FSA representing T_i, that is,
  $A_S = A_{T1} \| \ldots \| A_{Tn}$

Check Safety With Assertions

```
checkAssertions($A_S$) {
  seen := {$s_0$}
  stack := [$s_0$]
  DFS($s_0$)
}
```

- seen: set of states already explored
- stack: states on current path

```
DFS(s) {
  ws := enabled(s)
  for all a in ws {
    if a=assert(p) && !eval(p,s) then
      print("violation", s+stack)
    s' := execute(a, s)
    if s' not in seen {
      seen := seen + {s'}
      push(s', stack)
      DFS(s')
      pop(stack)
}}}
```

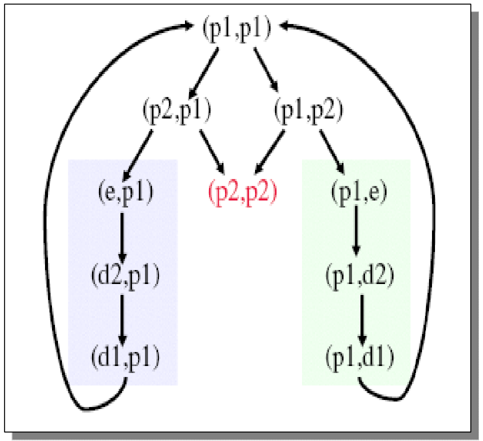
- ws := enabled(s):
  - get the transitions out of s (possibly "on-the-fly")
  - s records state of each thread Ti, i.e., s = (s_{T1}, …, s_{Tn})
- for all a in ws: pick one of the transitions to explore
- if a=assert(p) …: check for assertion violation, if necessary
- s' := execute(a, s): calculate the successor state
- if s' not in seen: if successor state has been seen before, ignore it
- DFS(s'): explore successor state

Check for deadlock is similar!

Source: 842@KSU

Check Safety With FSAs: Example

Let’s look at an example system [Phil1 || Phil2]:



```
boolean f1, f2;
thread Phil1() {
  loc pickup1: when !f1 do
                    {f1 := true;}
                 goto pickup2;
  loc pickup2: when !f2 do
                    {f2 := true;}
                 goto eating;
  loc eating:  do {}
                 goto drop2;
  loc drop2:   do {f2 := false;}
                 goto drop1;
  loc drop1:   do {f1 := false;}
                 goto pickup1;
}
```

Source: 842@KSU

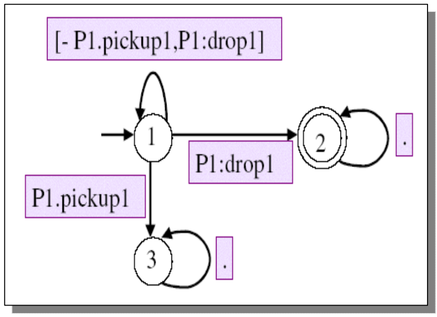
Check Safety With FSAs: Example (Cont’d)

... and an example property P_1:

“Phil1 must pickup Fork1 before dropping it”

$\neg P_1$ as reg. exp.: [- P1.pickup1, P1.drop1]*; P1.drop1; .*

$\neg P_1$ as a FSA:



$\neg P_1$ as a BIR program:

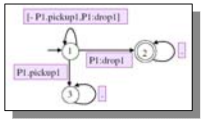
```
extension Property for ...
  {expdef boolean transform(string, string);}
function notP1() {
  loc init:
    when Property.transform(“Phil1”, ”drop1”)
      do {} goto bad;
    when Property.transform(“Phil1”, ”pickup1”)
      do {} goto good;
  loc good do {} goto good;
  loc bad do {} goto bad;
```

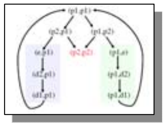
Source: 842@KSU

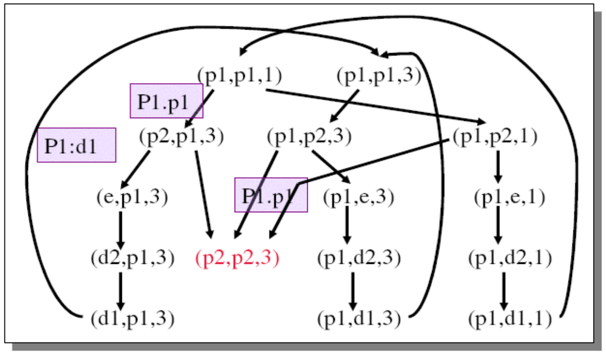
Check Safety With FSAs: Example (Cont’d)

To check whether [Phil1 || Phil2] satisfies P_1, we take the synchronous composition of [Phil1 || Phil2] and $\neg P_1$:

[Phil1||Phil2] ⊗ $\neg P_1$







Every word/execution ending in (*, *, 2) is

- in L([Phil1||Phil2] ⊗ $\neg P_1$)
- a violating execution!

Here:

- no violating executions
- system satisfies P_1!

Source: 842@KSU

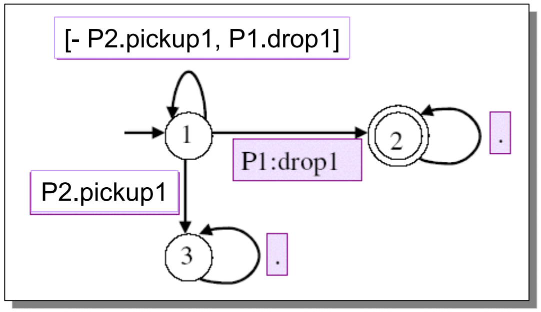
Check Safety With FSAs: Example (Cont’d)

Here’s another property P_2

“Phil2 must pickup Fork1 before Phil1 can drop it”

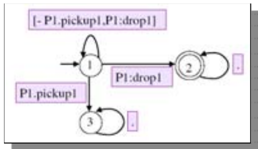
- $\neg P_2$ as regular expression:

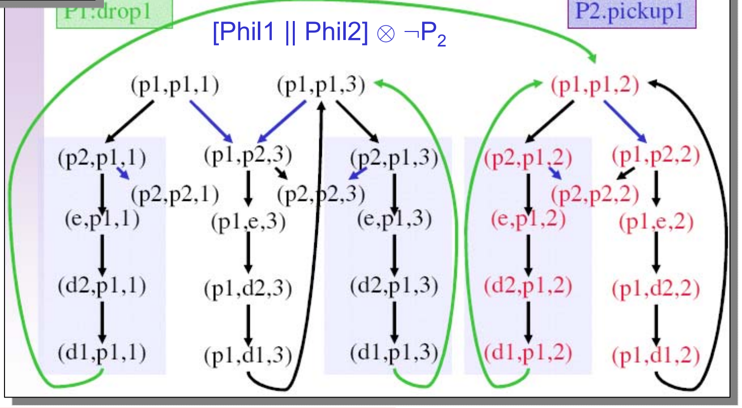
  [- P2.pickup1, P1.drop1]*; P1.drop1; .*
- $\neg P_2$ as FSA:



Source: 842@KSU

Check Safety With FSAs: Example (Cont'd)





Every word/execution ending in (*, *, 2) is

- in L([Phil1||Phil2] $\otimes \neg P_2$)
- a violating execution

Here:

- lots of violating executions
- system does not satisfy P_2

Source: 842@KSU

Check Safety With FSAs

- Let $A_S = A_{T_1} \| \ldots \| A_{T_n}$
- Let A_P be FSA expressing safety property P
- 3 Steps:
  1. build FSA $A_{\neg P}$ for negation of P
     - $A_{\neg P}$ must be total (use stutter extension)
  2. build $A_S \otimes A_{\neg P}$, synchronous product of A_S and $A_{\neg P}$
  3. do basic DFS on $A_S \otimes A_{\neg P}$
     - if a transition puts $A_{\neg P}$ into final state, then
       - Violation! Print contents of DFS stack as error trace
       - $L(A_S \otimes A_{\neg P})$ not empty
     - if $A_{\neg P}$ never reaches a final state, then
       - no violation! S satisfies P
       - $L(A_S \otimes A_{\neg P})$ empty

can be done at the same time ("on-the-fly" model checking, e.g., Spin, Bogor)

Check Safety With FSAs (Cont'd)

- Let A_S be $(S_S, s_{0,S}, L_S, \delta_S, F_S)$
- Let $A_{\neg P}$ be

  $(S_{\neg P}, s_{0,\neg P}, L_{\neg P}, \delta_{\neg P}, F_{\neg P})$

  where $A_{\neg P}$ is deterministic

```
checkSafety(A_S, A_¬P) {
  seen := {(s_0,S, s_0,¬P)}
  stack := [(s_0,S, s_0,¬P)]
  DFS((s_0,S, s_0,¬P))
}
```

new state component for $A_{\neg P}$

```
DFS((s, p)) {
  ws := enabled(s)
  for each a in ws {
    s' := execute(a, s)
    p' := (a in L_¬P) ? δ_¬P(p,a) : p
    if p' in F_¬P then
      print("violation", (s',p')+stack)
    if (s',p') not in seen then {
      seen := seen + {(s',p')}
      push((s',p'), stack)
      DFS((s',p'))
      pop(stack)
    }
  }
}
```

update state of $A_{\neg P}$

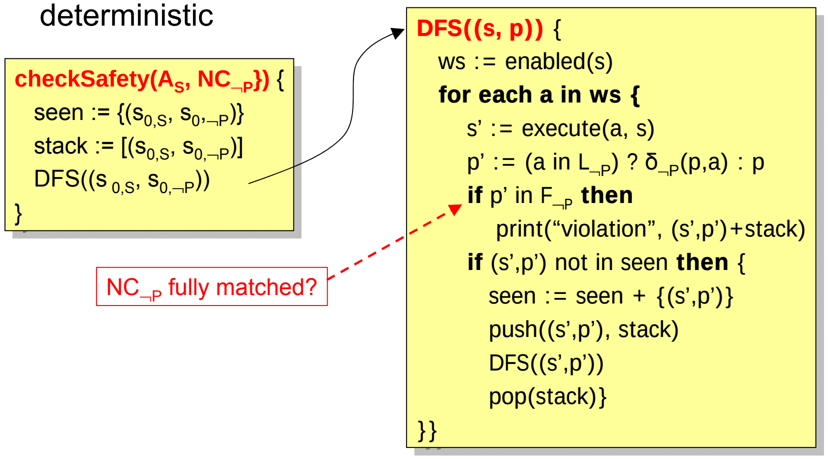
$A_{\neg P}$ in final state?

Check Safety With Never Claims

- In Spin, safety properties can be expressed using Never Claims
- Never Claims representing safety properties are FSAs
- Let $NC_{\neg P}$ be NC expressing negation of safety property P
- Check as before, except don't need to build negation
- 2 Steps:
  1. build $A_S \otimes NC_{\neg P}$, synchronous product of A_S and $NC_{\neg P}$
  2. do basic DFS on $A_S \otimes NC_{\neg P}$
     - final state of NC reached when NC "fully matched"
     - S violates P iff $L(A_S \otimes NC_{\neg P})$ not empty

Check Safety With Never Claims (Cont'd)

- Let A_S be $(S_S, s_{0,S}, L_S, \delta_S, F_S)$
- Let $NC_{\neg P}$ be $(S_{\neg P}, s_{0,\neg P}, L_{\neg P}, \delta_{\neg P}, F_{\neg P})$ where $NC_{\neg P}$ is deterministic



```
checkSafety($A_S$, $NC_{\neg P}$) {
  seen := {($s_{0,S}$, $s_{0,\neg P}$)}
  stack := [($s_{0,S}$, $s_{0,\neg P}$)]
  DFS(($s_{0,S}$, $s_{0,\neg P}$))
}
```

```
DFS((s, p)) {
  ws := enabled(s)
  for each a in ws {
    s' := execute(a, s)
    p' := (a in $L_{\neg P}$) ? $\delta_{\neg P}$(p,a) : p
    if p' in $F_{\neg P}$ then
      print("violation", (s',p')+stack)
    if (s',p') not in seen then {
      seen := seen + {(s',p')}
      push((s',p'), stack)
      DFS((s',p'))
      pop(stack)}
}}
```

$NC_{\neg P}$ fully matched?

Check Safety With LTL

- In Spin, safety properties can also be expressed as LTL properties
- Let P be safety property expressed in LTL
- Checking proceeds as before
- 3 Steps:
  1. build FSA $A_{\neg P}$ for negation of P
     - $A_{\neg P}$ must be total (use stutter extension)
  2. build $A_S \otimes A_{\neg P}$, synchronous product of A_S and $A_{\neg P}$
  3. do basic DFS on $A_S \otimes A_{\neg P}$
     - as before
     - S violates P iff $L(A_S \otimes A_{\neg P})$ not empty

---

Check Safety: In A Nutshell

- Let S be system with threads $T_1, \ldots, T_n$
- Let P be safety property
- Steps:
  1. build FSA $A_{\neg P}$ for negation of P
  2. build $A_S \otimes A_{\neg P}$, synchronous product of A_S and $A_{\neg P}$ where $A_S = A_{T1} \,||\, \ldots \,||\, A_{Tn}$, asynchronous composition of the T_i
  3. do basic DFS on $A_S \otimes A_{\neg P}$
- Complexity:
  - O(R) where R is # of reachable states in $A_S \otimes A_{\neg P}$

- Spin and Bogor do both steps at the same time ("on-the-fly")
- SMV carries steps out sequentially (not "on-the-fly")

Check Liveness With Never Claims

- Remember:
  - NC expresses violation of property
  - NC representing liveness property
    - = Buechi Automaton
    - = FSA + ω-acceptance
- Let
  - A_S be Buechi automaton representing the system S
  - $NC_{\neg P}$ express violation of liveness property P
  - t be execution of A_S
- Execution t violates P iff
  - "some 'good thing' never happens along t"

    iff t in $L^{\omega}(NC_{\neg P})$

    iff t causes $NC_{\neg P}$ into an 'acceptance cycle'

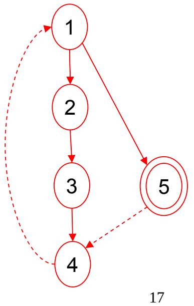
Check Liveness With Never Claims (Cont'd)

- S satisfies P

  iff $L^{\omega}(A_S \otimes NC_{\neg P})$ is empty

  iff $A_S \otimes NC_{\neg P}$ has no accepting execution

  iff $A_S \otimes NC_{\neg P}$ has no execution that ends in an accepting cycle
- To check if S satisfies P
  1. build $A_S \otimes NC_{\neg P}$
  2. check if $A_S \otimes NC_{\neg P}$ has acceptance cycle
     - How to do that?
     - Basic DFS or BFS is not enough…



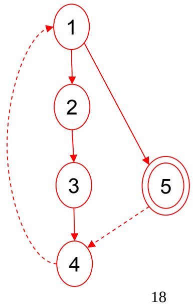
Check Liveness With Never Claims (Cont'd)

- Solution 1:
  - Compute strongly connected components (SCC) in $A_S \otimes NC_{\neg P}$ (Tarjan's algorithm)
  - $A_S \otimes NC_{\neg P}$ has acceptance cycle iff
    - $A_S \otimes NC_{\neg P}$ has SCC such that
      - SCC reachable from initial state, and
      - SCC contains at least one accepting state
- Solution 2: (easier)
  - Check if $A_S \otimes NC_{\neg P}$ has at least one state s s.t.

    (1) s is accepting

    (2) s reachable from initial state

    (3) s is reachable from itself
  - Implementation: Nested DFS
    - First DFS to find s s.t. (1) and (2)
    - Then, nested DFS to check (3)



Check Liveness With Never Claims (Cont'd)

- Let A_S be $(S_S, s_{0,S}, L_S, \delta_S, F_S)$
- Let $NC_{\neg P}$ be

  $(S_{\neg P}, s_{0,\neg P}, L_{\neg P}, \delta_{\neg P}, F_{\neg P})$

  where $NC_{\neg P}$ is deterministic

```
checkLiveness(A_S, NC_¬P}) {
  seen1 := {(s_0,S, s_0,¬P)}
  stack1 := [(s_0,S, s_0,¬P)]
  DFS((s_0,S, s_0,¬P))
}
```

```
DFS((s, p)) {
  ws1 := enabled(s)
  for each a in ws {
    s' := execute(a, s)
    p' := (a in L_¬P) ? δ_¬P(p,a) : p
    if (s',p') not in seen then {
      seen1 := seen1 + {(s',p')}
      push((s',p'), stack1)
      DFS((s',p'))
      if p' in F_¬P then {
        seen2 = {(s',p')}
        stack2 = [(s',p')]
        NDFS((s',p'), (s',p'))}
      pop(stack1)
}}}
```

is p' accepting state?

start nested DFS if p' is an accepting state of $NC_{\neg P}$

Check Liveness With Never Claims (Cont'd)

```
DFS((s, p)) {
  ws1 := enabled(s)
  for each a in ws {
    s' := execute(a, s)
    p' := (a in L_¬P) ? δ_¬P(p,a) : p
    if (s',p') not in seen then {
      seen1 := seen1 + {(s',p')}
      push((s',p'), stack1)
      DFS((s',p'))
      if p' in F_¬P then {
        seen2 = {(s',p')}
        stack2 = [(s',p')]
        NDFS((s',p'), (s',p')) }
      pop(stack1)}
}}
```

```
NDFS((s, p), start) {
  ws2 := enabled(s)
  for each a in ws2 {
    s' := execute(a, s)
    p' := (a in L_¬P) ? δ_¬P(p,a) : p
    if (s',p') = start then
      print("violation",
              stack1+stack2)
    if (s',p') not in seen2 then {
      seen2 := seen2 + {(s',p')}
      push((s',p'), stack2)
      NDFS((s',p'), start)
      pop(stack2)}
}}
```

acceptance cycle found!

Check Liveness With Never Claims (Cont'd)

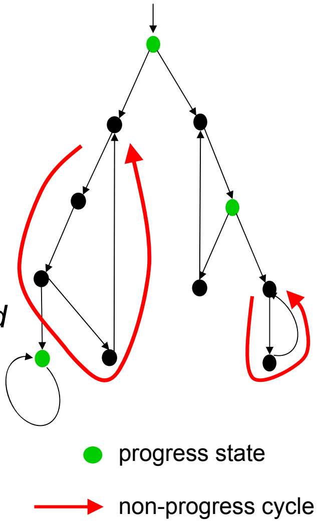
- Let $NC_{\neg P}$ be NC expressing negation of liveness property P
- 2 Steps:
  1. build $A_S \otimes NC_{\neg P}$, synchronous product of A_S and $NC_{\neg P}$
  2. do nested DFS on $A_S \otimes NC_{\neg P}$ to search for acceptance cycle
     - S violates P iff
       - $A_S \otimes NC_{\neg P}$ has acceptance cycle
       - $L^{\omega}(A_S \otimes NC_{\neg P})$ not empty

Check Liveness With LTL

- Let P be an LTL formula expressing a liveness property
- Build $NC_{\neg P}$ representing negation of P
- Then, as before
- 3 Steps:
  1. build $NC_{\neg P}$, never claim representing ¬P
  2. build $A_S \otimes NC_{\neg P}$, synchronous product of A_S and $NC_{\neg P}$
  3. do nested DFS on $A_S \otimes NC_{\neg P}$ to search for acceptance cycle
     - S violates P iff
       - $A_S \otimes NC_{\neg P}$ has acceptance cycle
       - $L^{\omega}(A_S \otimes NC_{\neg P})$ not empty

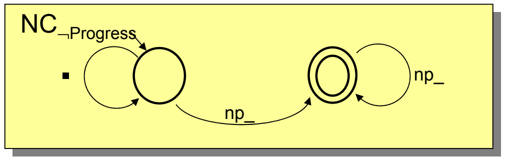
Check Liveness With Progress Labels

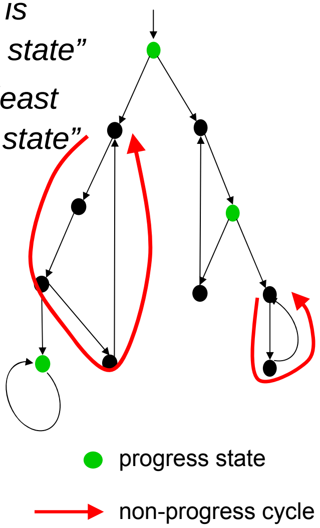
- Need to find non-progress cycles
- Remember: In Spin,
  - whether or not the system makes progress in a given state s is observable
  - **np_** false in s iff at least one process is at progress label in s
- Let *Progress* be *"every state along every path is always eventually followed by a progress state"*
- Idea: Use **np_** to express *Progress* and *¬Progress* as LTL formulas
- Which?



Check Liveness With Progress Labels (Cont'd)

- Progress = *"every state along every path is always eventually followed by a progress state"*
- ¬Progress = *"at least one state along at least one path is never followed by a progress state"*
- In LTL using **np_**:
  - Progress = []<> ¬**np_**
  - ¬Progress = <>[]**np_**
- As Never Claim $NC_{\neg Progress}$:





Check Liveness With Progress Labels (Cont'd)

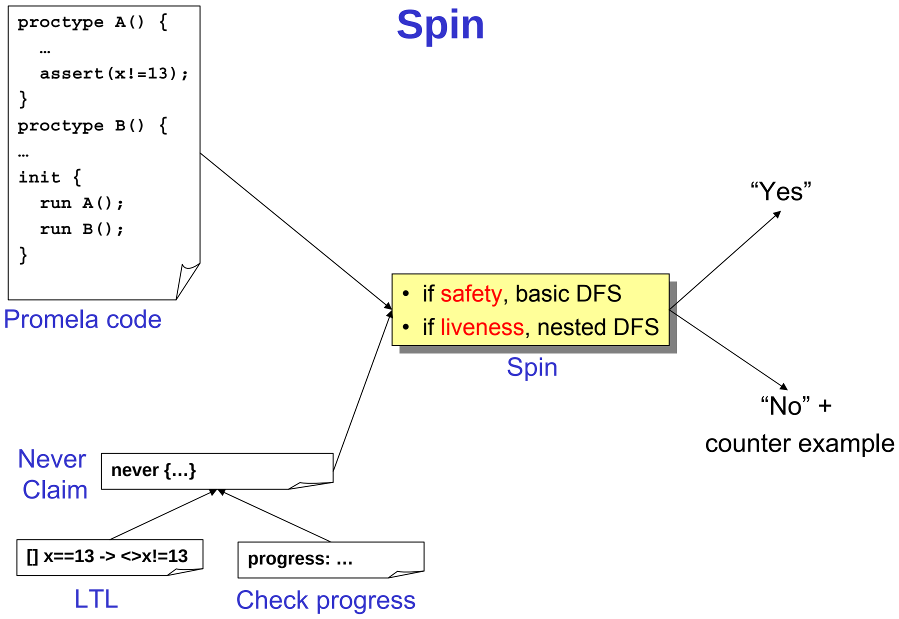
- 3 Steps:
  1. build $NC_{\neg Progress}$, never claim representing non-progress
  2. build $A_S \otimes NC_{\neg Progress}$, synchronous product of A_S and $NC_{\neg Progress}$
  3. do nested DFS on $A_S \otimes NC_{\neg Progress}$ to search for non-progress cycle
     - S violates Progress iff
       - $A_S \otimes NC_{\neg P}$ has acceptance cycle
       - $L^{\omega}(A_S \otimes NC_{\neg P})$ not empty

Check ~~Safety~~ Liveness: In A Nutshell

- Let S be system with threads $T_1, \ldots, T_n$
- Let P be ~~safety~~ liveness property
- Steps:
  1. build ~~FSA~~ Buechi Automaton $A_{\neg P}$ for negation of P
  2. build $A_S \otimes A_{\neg P}$, synchronous product of A_S and $A_{\neg P}$ where $A_S = A_{T1} \| \ldots \| A_{Tn}$, asynchronous composition of the T_i
  3. do ~~basic~~ nested DFS on $A_S \otimes A_{\neg P}$

  - Spin and Bogor do both steps at same time ("on-the-fly")
  - SMV carries steps out sequentially
- Complexity:
  - ~~O(R)~~ O(2·R) where R is # of reachable states in $A_S \otimes A_{\neg P}$

Spin



```
proctype A() {
  …
  assert(x!=13);
}
proctype B() {
…
init {
  run A();
  run B();
}
```

Promela code

Never Claim

never {…}

[] x==13 -> <>x!=13

LTL

progress: …

Check progress

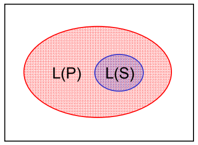
- if safety, basic DFS
- if liveness, nested DFS

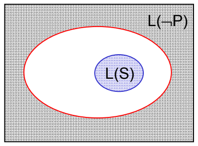
Spin

"Yes"

"No" + counter example

The Language-Theoretic View





- L(S) : system executions
- L(P) : executions satisfying the property
- Need to determine: $L(S) \subseteq L(P)$
- Observation: $A \subseteq B$ iff $(A \cap \neg B) = \emptyset$
- So, to see if $L(S) \subseteq L(P)$, we
  - Step 1: take ¬P
  - Step 2: see if $L(S) \cap L(\neg P)$ is empty, that is, if there does not exist an execution t such that
    - S can do t, that is, t in L(S), and
    - t violates P, that is, t is in L(¬P)
  - Step 2 will succeed precisely when $S \otimes \neg P$ has no accepting executions
- Theorem: Buechi Automata are closed under negation, union and intersection

Important Properties of Buechi-Automata

Buechi automata are closed under complement, union, and intersection.

Let A and B be Buechi-automata. Then,

- **¬A** denotes the automaton that accepts precisely the words not accepted by A:

$$L^{\omega}(\neg A) = \{w \mid w \notin L^{\omega}(A)\}$$

- **A∪B** denotes the automaton that accepts precisely the words accepted by A or by B:

$$L^{\omega}(A \cup B) = L^{\omega}(A) \cup L^{\omega}(B)$$

- Similarly for **A∩B**

Weak Fairness

- So far, it's possible that along a counter example a process stops moving although it is enabled

⇒ such counter examples are not very realistic (why?)

Definition: An ω-run σ satisfies the *weak fairness* requirement if it contains infinitely many transitions from every process (component automaton in the asynchronous product) that is *enabled* (has an executable action) *infinitely long* in σ

- Nested DFS can be adapted to enforce fairness (more details in Spin book Chapter 8)
- Cost: linear increase in complexity (in # of processes)

Complexity and Optimization

- Size of $A_S \otimes A_{\neg P}$
  - R = # of reachable states in $A_S \otimes A_{\neg P}$
  - $R = R_S \cdot R_{\neg P}$ where
    - R_S = # of reachable states in A_S (typically: $10^9 \ldots 10^{11}$)
    - $R_{\neg P}$ = # of reachable states in $A_{\neg P}$ (typically: 1..4)
- Size of A_S
  - $R_S = R_{T1} \cdot \ldots \cdot R_{Tn} \sim R_T^n$
- Size of T
  - $R_T = (\text{\# loc's in T}) \cdot |dtype_1| \cdot \ldots \cdot |dtype_m| \sim (\text{\# loc's in T}) \cdot |dtype|^m$
- Thus,
  - $R_S = \left((\text{\# loc's in T}) \cdot |dtype|^m \right)^n$

R_S increases with
- # of processes n (exponentially)
- # of variables m
- size of data types
- size of process

Complexity and Optimization (Cont'd)

- Size of $A_S \otimes A_{\neg P}$
  - $R = R_S \cdot R_{\neg P} = \left((\text{\# loc's in T}) \cdot |dtype|^m \right)^n \cdot R_{\neg P}$
- Reduce R by

reducing
- # of processes n (exponentially)
- # of variables m
- size of data type dtype
- size of process T
- size of specification P

user

using
- partial order reduction
- statement merging
- abstraction

checker/user

- Reduce memory requirement by
  - compression

CTL Model Checking Algorithm (1)

- So much for LTL model checking
- Now, on to CTL model checking
- Algorithm quite different, because CTL quite different from LTL

CTL Model Checking Algorithm (2)

Definition: A set of connectives S is adequate for CTL iff for every CTL formula φ, there exists an equivalent CTL formula T(φ) that only contains the connectives in S

Theorem: {¬, ∨, EX, AF, EU} is adequate for CTL

Proof:

$\varphi_1 \wedge \varphi_2 \leftrightarrow \neg(\neg\varphi_1 \vee \neg\varphi_2)$

$\varphi_1 \rightarrow \varphi_2 \leftrightarrow \neg\varphi_1 \vee \varphi_2$

$AX\varphi \leftrightarrow \neg EX\neg\varphi$

$AG\varphi \leftrightarrow \neg EF\neg\varphi$

$EG\varphi \leftrightarrow \neg AF\neg\varphi$

$EF\varphi \leftrightarrow E[tt\ U\ \varphi]$

$A[\varphi_1\ U\ \varphi_2] \leftrightarrow \neg(EG\neg\varphi_2 \vee E[\neg\varphi_2\ U\ \neg\varphi_1 \wedge \varphi_2])$

$\leftrightarrow AF\varphi_2 \wedge \neg E[\neg\varphi_2\ U\ \neg\varphi_1 \wedge \varphi_2]$

CTL Model Checking Algorithm (3)

- Recall
  - $AG\varphi \leftrightarrow \varphi \wedge AX\ AG\varphi$
  - $EG\varphi \leftrightarrow \varphi \wedge EX\ EG\varphi$
  - $AF\varphi \leftrightarrow \varphi \vee AX\ AF\varphi$
  - $EF\varphi \leftrightarrow \varphi \vee EX\ EF\varphi$
  - $A[\varphi_1\ U\ \varphi_2] \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge AX\ A[\varphi_1\ U\ \varphi_2])$
  - $E[\varphi_1\ U\ \varphi_2] \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge EX\ E[\varphi_1\ U\ \varphi_2])$

CTL Model Checking Algorithm (4)

Input: FSM $M=(S, s_0, L, \rightarrow, F)$ and CTL formula φ over AP
Output: "yes" if $M \models \varphi$, "no" otherwise

Step 0: Let φ' be T(φ)

Step 1: For all subformulas ψ in φ' (starting w/ smallest) including φ', label all states s in M satisfying ψ:

```
Sat($\psi$) = CASE $\psi$ OF
  $p \in AP$ : label a state s w/ p if p true in s
  $\neg\psi'$ : Sat($\psi$); label a state s w/ $\neg\psi$ if s is not labeled w/ $\psi$
  $\psi_1 \vee \psi_2$: Sat($\psi_1$); Sat($\psi_2$); label a state s w/ $\psi_1 \vee \psi_2$ if s labeled w/ $\psi_1$ or $\psi_2$
  EX $\psi'$: Sat($\psi'$); label a state s w/ EX$\psi'$ if at least one successor of s is labeled w/ $\psi'$
  AF$\psi'$: Sat($\psi'$);
      Repeat
          label state s w/ AF$\psi'$ if s labeled w/ $\psi'$ or all successors of s labeled w/ AF$\psi'$
      Until labeling doesn't change anymore, i.e., a "fixed point" is reached
  E[$\psi_1$ U $\psi_2$] : Sat($\psi_1$); Sat($\psi_2$);
          Repeat
              label state s w/ E[$\psi_1$U $\psi_2$] if s labeled w/ $\psi_2$ or (s labeled w/ $\psi_1$
                      and at least one successor of s labeled w/ E[$\psi_1$ U $\psi_2$])
          Until labeling doesn't change anymore, i.e., a "fixed point" is reached
```

CTL Model Checking Algorithm (5)

Input: FSM M=(S, s_0, L, →, F) and CTL formula φ over AP

Output: "yes" if M⊨φ, "no" otherwise

Step 0: Let φ' be T(φ)

Step 1: For all subformulas ψ in φ' (starting w/ smallest) including φ', label all states s in M satisfying ψ:

Step 2: If s_0 labeled with φ, then output "yes", else output "no"

Example:

M⊨AF p?

"yes"

Complexity: O((|S|+|→|) · |φ|)

LTL Model Checking vs. CTL Model Checking

To check M ⊨ φ

LTL model checking:

1) Check if $L(M \otimes A_{\neg\varphi}) = \emptyset$ where $A_{\neg\varphi}$ is non-deterministic Buechi Automaton representing φ
2) Check implemented by
   a) for safety: DFS or BFS
   b) for liveness: nested DFS
3) Note
   a) execution sequences are linear (non-branching)
   b) transition relation of M can be computed "on-the-fly"
   c) In worst case, $|A_{\neg\varphi}|$ exponential in |φ|
4) Complexity: $O(|M| \cdot 2^{|\varphi|})$, but |M| dominates $2^{|\varphi|}$ in practice
5) Sample tools: Spin, Bogor, JPF

CTL model checking:

1) Check if (M,s)⊨φ for all $s_0 \in M.S_0$
2) Check implemented by
   a) express φ in terms of {¬, ∨, EX, AF, EU}
   b) labeling algorithm Sat(φ) that is inductive over structure of φ and uses fixed point computation
3) Note
   a) execution sequences are branching
   b) transition relation of M cannot be computed on-the-fly
4) Complexity: O(|M|· |φ|)
5) Sample tools: SMV

Projects and Presentations

Schedule

Now:	pick project
Till week of April 6:	work on project
Week of April 6:	presentations & summary papers

Presentations

- 20 mins
- group members take turns

Summary papers

- b/w 2-5 pages in ACM SIG Proceedings format
- to be distributed at presentation time