Design and Analysis of Distributed Interacting Systems

Lecture 6 – LTL Model Checking

Prof. Dr. Joel Greenyer

May 16, 2013
Some Book References (1)


Some Book References (2)


Some Book References (3)


Some Book References (4)


mtype = {press, hold};

chan c = [0] of { mtype };

active proctype switch()
{
    RELEASED:
        if
            :: c!press; goto PRESSED
        fi;
    PRESSED:
        if
            :: c!hold; goto PRESSED
            :: goto RELEASED
        fi;
}

active proctype light()
{
    OFF:
        if
            :: c?press; goto LOW
        fi;
    LOW:
        if
            :: c?press; goto OFF
            :: c?hold; goto HIGH
        fi;
    HIGH:
        if
            :: c?press; goto OFF
            :: c?hold; goto LOW
        fi;
}

(this is a possible pattern to model state machines in Promela)
Spin Verification more Technically...

```c
1  byte x, y;
2  active proctype mini(){
3      do
4          :: (x < 2) ->
5              x++
6          :: (y < 2) ->
7              y++
8          :: else ->
9              break
10     od
11  }
```

Promela model

C program

Spin settings

Output

-4: -4: -4
1: 1: 17
2: 1: 23
3: 0: 0
4: 1: 17
5: 0: 4
6: 1: 21
7: 1: 23

Error Trace

verification result:
spin -a light_brightness.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAF
/pn -m10000
Pid: 9028
pan: 1: invalid end state (at depth 6)
pan: wrote light_brightness.pml.traj

(Spin Version 6.2.3 -- 24 October 2012)
Warning: Search not completed
+ Partial Order Reduction
Spin Models and Kripke Structures

- A Spin model can be translated to a Kripke Structure
  - data types, channels, max. no. of processes is finite
  - Spin can do an exhaustive analysis of the corresponding KS
  - Spin constructs KS “on-the-fly”, i.e., sometimes it finds results without constructing the complete KS

```c
byte x, y;
active proctype mini(){
    do
        :: (x < 2) -> x++
        :: (y < 2) -> y++
        :: else -> break
    od
}
```

```
(_, 3, 0, 0)
(0, 5, 0, 0)   (0, 7, 0, 0)
(0, 3, 1, 0)   (0, 3, 0, 1)
(0, 5, 0, 1)   (0, 7, 0, 1)
...             ...
```

x<2   y<2
x++   y++
...   ...
...   ...
 Assertions

... #define trainOnCrossing 3 #define carOnCrossing 2 ...

active proctype train(){
  byte state;
  ...
}

active proctype car(){
  byte state;
  ...
}

active proctype Inv(){
  assert(!(train:state == trainOnCrossing &&
           car:state == carOnCrossing))
}

during the exhaustive state space exploration during model checking, all possible interleavings of the other processes and executing this assertion will be checked

when is this assertion executed?
**Verify LTL Properties**

\[
\text{mtype} = \{\text{press, hold}\};
\]

\[
\text{chan } c = [0] \text{ of } \{ \text{mtype} \};
\]

\[
\text{active proctype switch()}{
\quad \text{RELEASED:}
\qquad \text{if}
\qquad \quad :: c!\text{press}; \text{goto PRESSED}
\qquad \text{fi;}
\]

\[
\quad \text{PRESSED:}
\qquad \text{if}
\qquad \quad :: c!\text{hold}; \text{goto PRESSED}
\qquad :: \text{goto RELEASED}
\qquad \text{fi;}
\]

\[
\text{active proctype light()}{
\quad \text{OFF:}
\qquad \text{if}
\qquad \quad :: c?\text{press}; \text{goto LOW}
\qquad \text{fi;}
\quad \text{LOW:}
\qquad \text{if}
\qquad \quad :: c?\text{press}; \text{goto OFF}
\qquad :: c?\text{hold}; \text{goto HIGH}
\qquad \text{fi;}
\quad \text{HIGH:}
\qquad \text{if}
\qquad \quad :: c?\text{press}; \text{goto OFF}
\qquad :: c?\text{hold}; \text{goto LOW}
\qquad \text{fi;}
\}
\]

[] stands for G (always),
<> stands for F (eventually),
! is ¬

\[
\text{ltl p0} \{[]<> \text{light@LOW}\}
\]

\[
\text{ltl p1} \{[]<> \text{light@HIGH}\}
\]
Never-Claim

• Sequence of Boolean expressions over variables in the model that must never happen

• Simple example:

```java
byte x = 3;

active proctype P(){
    x = 1;
}

never{
    x == 3;
    x == 1
}
```

The never-claim reaches its end and the verification will thus report a violation.
... never { true; light@LOW; true; light@HIGH; }

I didn't explain this thoroughly.
Never-Claims – Why should I bother?

• If I know how to specify interesting properties in LTL, why should I bother “programming” never-claims?

• Spin checks LTL properties by first converting them to never-claims
  – understanding never-claims helps understanding how Spin checks LTL properties

• Never-claims are more verbose than LTL formulae, but they are also more powerful
  – e.g., you can count (up to finite numbers) in never-claims
The Never Claim Checking Process

- Initializing global variables
- Initialize init and active processes
- Initialize local process variables
- [never-claim can make step] → never-claim makes a step → [model can make step]
- [model cannot make step] → no counter-example
- [never-claim cannot make step] → no counter-example
- [never-claim cannot make step] → [accept-cycle found]
- model makes a step → [model can make step]
- [model cannot make step]
- counter-example found
- [counter-example found] → [accept-cycle found]
- [accept-cycle found] → no counter-example
- [never-claim terminated] → counter-example found
- [never-claim not terminated] → counter-example found

active proctype light()

OFF:
 if :: c?press; goto LOW
 fi;

LOW:
 if :: c?press; goto OFF
 :: c?hold; goto HIGH
 fi;

HIGH:
 if :: c?press; goto OFF
 :: c?hold; goto LOW
 fi;

never {
  true;
  light@LOW;
  true;
  light@HIGH;
}

1. true can always make a step (light@OFF)
2. model makes step (light@LOW)
3. Can now make this step
4. model makes step (switch@RELEASED, not shown here)
5. Makes step
6. model makes step (light@HIGH)
Another Never-Claim Example

• Labels of the form `accept[a-zA-Z0-9_]` mark acceptance cycles
  – it must not be possible to visit these labels infinitely often

• What is specified here?

```plaintext
T0_init:
  if
    :: (! light@LOW) -> goto accept_S4
    :: true -> goto T0_init
  fi;
accept_S4:
  if
    :: (! light@LOW) -> goto accept_S4
  fi;
```

Non-determinisit choice – remember: all possibilities will be explored during a verification run.
Design and Analysis of Distributed Interacting Systems

Lecture 6 – LTL Model Checking

Prof. Dr. Joel Greenyer

May 16, 2013
Model-Checking

Model (Kripke structure)

Specification (LTL)

Model Checking

modify

+ counter example
(how the specification can be violated)

false

ture
Automata-based LTL Model Checking

• There are different techniques for checking LTL properties
  – i.e. checking whether $M \models \varphi$

• One is based on Büchi Automata (BA)
  – automata that accept infinite words

• Approach: (Be $M$ a Kripke structure over $AP$)

$M \models \varphi$
iff $L(M) \subseteq L(\varphi)$
iff $L(M) \cap ((2^{AP})^{\omega} \setminus L(\varphi)) = \emptyset$
iff $L(M) \cap L(\neg \varphi) = \emptyset$
iff $L(B_M \otimes B_{\neg \varphi}) = \emptyset$

What we need:
1. Checking emptyness of the language accepted by a BA
2. Product construction for BAs
3. Represent KS as BA
4. Represent LTL formula as BA
1. Introduce Büchi Automata
2. Checking emptiness of the language accepted by a BA
3. Product construction for BAs
4. Represent KS as BA
5. Represent LTL formula as BA
Büchi Automata

• A Büchi automaton is a tuple $BA = (Q, \Sigma, T, I, F)$ in which
  – $Q$ is a finite, non-empty set of states
  – $\Sigma$ is a finite alphabet
  – $T \subseteq Q \times \Sigma \times Q$ is a transition relation
  – $I \subseteq Q$ is a set of initial state
  – $F \subseteq Q$ is a set of final states (also called accepting states)

• An infinite word $\pi \in \Sigma^\omega$ is accepted by a BA iff the BA has a corresponding run (a path starting from an initial state), that infinitely often visits final states.
  – Such a run is also called an accepting run

• Example:
 infinitely often $a$,
  $\Sigma = \{a, b\}$
Büchi Automata

- There are other kinds of automata for infinite words
  - Rabin automata
  - Muller automata
  - Street automata
- they all accept the class of $\omega$-regular languages

- Note: Not all languages accepted by a non-deterministic Büchi automaton are accepted by a deterministic one
  - example: words with finitely many $a$s
  - cannot be represented by a deterministic BA

\[
\text{Diagram of Büchi Automaton:}
\]
Agenda

1. Introduce Büchi Automata (√)
2. Checking emptiness of the language accepted by a BA
3. Product construction for BAs
4. Represent KS as BA
5. Represent LTL formula as BA
Checking Emptyness

• An accepting run must visit at least one accepting state infinitely often
• How do we determine the existence of an accepting run?
• An accepting state must thus appear in a cycle reachable from a start state.
Find Accepting Runs – SCC-Based Approach

• Compute all *strongly connected components* (SCCs)
• Check whether a non-trivial SCC contains an accepting state and whether it is reachable from a start state

• Def.: States $C \subseteq Q$ form a *strongly connected component* iff
  – for all $q, q' \in C$: $q$ is reachable from $q'$
  – There is no $C' \subseteq C$ for which this is true ($C$ is maximal)

• An SCC is trivial iff $|C| = 1$ and for $q \in C$: $(q, \sigma, q) \notin T$, $\sigma \in \Sigma$
  – non-trivial SCCs?

Tarjan's algorithm, linear in the size of the graph, reachability as well, thus overall: $O(|Q| + |T|)$
Find Accepting Runs – Another Idea: DFS

• Start a depth first search from an initial state of the BA
  – remember DFS: uses a stack for backtracking
• when from a state $q$ an edge is found to a state $q'$ that is currently on the stack, a cycle is found
  – the cycle is along the states on the stack from of $q'$ to $q$
  – If one of these states is accepting, there is an accepting run
    • along the state on the stack, and then repeating in the cycle

• An edge to a state on the stack is called a backward edge
  – If DFS finds no backward edges, then the BA is acyclic

• Problem: When we find a cycle, we must always check if it contains an accepting state …this is expensive, we are not anymore linear in the size of the automaton.
Find Accepting Runs – Nested DFS

• Idea:
  – Two DFSs, called blue (outer) and red (inner) DFSs
  – each DFS visits a state at most once, coloring it blue/red

  – Start blue DFS from a start state
  – if blue DFS finds an accepting state $q$, start red DFS from $q$
  – if red DFS finds a non-empty path from $q$ to $q$, report a cycle, an accepting run is found: current stack of blue DFS + cycle
  – otherwise continue blue DFS (from $q$)
procedure nested_dfs(BA a) 
    forall $q_0 \in I_a$ call dfs_blue($q_0$); 

procedure dfs_blue (State $q$) 
    $q$.blue := true; 
    forall $q' \in \text{post}(q)$ do 
        if $\neg q'.blue$ then 
            call dfs_blue($q'$); 
        else if $q' = \text{seed}$ then 
            report cycle; 

procedure dfs_red (State $q$) 
    $q$.red := true; 
    forall $q' \in \text{post}(q)$ do 
        if $\neg q'.red$ then 
            call dfs_red($q'$); 
        else if $q' = \text{seed}$ then 
            report cycle;
procedure nested_dfs(BA a)
    forall q₀ ∈ I_a call dfs_blue(q₀);

procedure dfs_blue (State q)
    q.blue := true;
    forall q' ∈ post(q) do
        if ¬q'.blue then call dfs_blue(q');
        if q ∈ F_a then
            seed := q;
            call dfs_red(q);

procedure dfs_red (State q)
    q.red := true;
    forall q' ∈ post(q) do
        if ¬q'.red then call dfs_red(q');
        else if q' = seed then report cycle;
procedure nested_dfs(BA \( a \))
  forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
  \( q.blue := \) true;
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.blue \) then
      call dfs_blue(\( q' \));
    if \( q \in F_a \) then
      \( \text{seed} := q; \)
      call dfs_red(\( q \));

procedure dfs_red (State \( q \))
  \( q.red := \) true;
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.red \) then
      call dfs_red(\( q' \));
    else if \( q' = \text{seed} \) then
      report cycle;
**Find Accepting Runs – Nested DFS**

**procedure** nested_dfs(BA a)  
  forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

**procedure** dfs_blue(State \( q \))  
  \( q.blue := \text{true}; \)
  forall \( q' \in \text{post}(q) \) do  
    if \( \neg q'.blue \) then  
      call dfs_blue(\( q' \));
    else if \( q' = \text{seed} \) then  
      report cycle;

**procedure** dfs_red(State \( q \))  
  \( q.red := \text{true}; \)
  forall \( q' \in \text{post}(q) \) do  
    if \( \neg q'.red \) then  
      call dfs_red(\( q' \));
    else if \( q' = \text{seed} \) then  
      report cycle;
procedure nested_dfs(BA a)
  forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
  \( q.blue := \text{true} \);
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.blue \) then
      call dfs_blue(\( q' \));
    if \( q \in F_a \) then
      seed := q;
      call dfs_red(q);

procedure dfs_red (State \( q \))
  \( q.red := \text{true} \);
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.red \) then
      call dfs_red(\( q' \));
    else if \( q' = \text{seed} \) then
      report cycle;
procedure nested_dfs(BA a)
    forall $q_0 \in I_a$ call dfs_blue($q_0$);

procedure dfs_blue (State $q$)
    $q$.blue := true;
    forall $q' \in \text{post}(q)$ do
        if $\neg q'.blue$ then
            call dfs_blue($q'$);
        if $q \in F_a$ then
            seed := $q$;
            call dfs_red($q$);

procedure dfs_red (State $q$)
    $q$.red := true;
    forall $q' \in \text{post}(q)$ do
        if $\neg q'.red$ then
            call dfs_red($q'$);
        else if $q' = \text{seed}$ then
            report cycle;

---

Find Accepting Runs – Nested DFS

---

Diagram showing a graph with states and transitions, illustrating the process of finding accepting runs using nested DFS.
Find Accepting Runs – Nested DFS

**procedure** nested_dfs(BA \(a\))

\[
\text{forall } q_0 \in I_a \text{ call dfs_blue}(q_0);
\]

**procedure** dfs_blue (State \(q\))

\[
q.blue := \text{true};
\]

\[
\text{forall } q' \in \text{post}(q) \text{ do}
\]

\[
\text{if } \neg q'.blue \text{ then}
\]

\[
\text{call dfs_blue}(q');
\]

\[
\text{if } q \in F_a \text{ then}
\]

\[
\text{seed} := q;
\]

\[
\text{call dfs_red}(q);
\]

**procedure** dfs_red (State \(q\))

\[
q.red := \text{true};
\]

\[
\text{forall } q' \in \text{post}(q) \text{ do}
\]

\[
\text{if } \neg q'.red \text{ then}
\]

\[
\text{call dfs_red}(q');
\]

\[
\text{else if } q' = \text{seed} \text{ then}
\]

\[
\text{report cycle;}
\]
procedure nested_dfs(BA a)
    forall $q_0 \in I$ call dfs_blue($q_0$);

procedure dfs_blue (State $q$)
    $q$.blue := true;
    forall $q' \in \text{post}(q)$ do
        if $\neg q'.blue$ then.
            call dfs_blue($q'$);
        else if $q' = \text{seed}$ then
            report cycle;

procedure dfs_red (State $q$)
    $q$.red := true;
    forall $q' \in \text{post}(q)$ do
        if $\neg q'.red$ then
            call dfs_red($q'$);
        else if $q' = \text{seed}$ then
            report cycle;
procedure nested_dfs(BA a)
   forall $q_0 \in I_a$ call dfs_blue($q_0$);

procedure dfs_blue (State $q$)
   $q$.blue := true;
   forall $q' \in \text{post}(q)$ do
      if $\neg q'.$blue then
         call dfs_blue($q'$);
      else if $q' = \text{seed}$ then
         report cycle;

procedure dfs_red (State $q$)
   $q$.red := true;
   forall $q' \in \text{post}(q)$ do
      if $\neg q'.$red then
         call dfs_red($q'$);
      else if $q' = \text{seed}$ then
         report cycle;
Find Accepting Runs – Nested DFS

```plaintext
procedure nested_dfs(BA a)
    forall q₀ ∈ Iₐ call dfs_blue(q₀);

procedure dfs_blue (State q)
    q.blue := true;
    forall q' ∈ post(q) do
        if q'.blue then
            call dfs_blue(q');
        if q ∈ Fa then
            seed := q;
            call dfs_red(q);

procedure dfs_red (State q)
    q.red := true;
    forall q' ∈ post(q) do
        if q'.red then
            call dfs_red(q');
        else if q' = seed then
            report cycle;
```

Diagram of a state transition diagram with nodes and edges indicating the traversal order and conditions for accepting runs.
Find Accepting Runs – Nested DFS

procedure nested_dfs(BA a)
  forall \( q_0 \in I_a \) call dfs_blue\((q_0)\);

procedure dfs_blue (State \( q \))
  \( q.blue := true \);
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.blue \) then
      call dfs_blue\((q')\);
    if \( q \in F_a \) then
      seed := \( q \);
      call dfs_red\((q)\);

procedure dfs_red (State \( q \))
  \( q.red := true \);
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'.red \) then
      call dfs_red\((q')\);
    else if \( q' = \text{seed} \) then
      report cycle;

Find Accepting Runs – Nested DFS

**procedure** nested_dfs(BA \( a \))

forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

**procedure** dfs_blue (State \( q \))

\( q.blue := \text{true} \);

forall \( q' \in \text{post}(q) \) do

if \( \neg q'.blue \) then

call dfs_blue(\( q' \));

if \( q \in F_a \) then

seed := \( q \);

call dfs_red(\( q \));

**procedure** dfs_red (State \( q \))

\( q.red := \text{true} \);

forall \( q' \in \text{post}(q) \) do

if \( \neg q'.red \) then

call dfs_red(\( q' \));

else if \( q' = \text{seed} \) then

report cycle;

\( \text{seed} \)
Find Accepting Runs – Nested DFS

```
procedure nested_dfs(BA a)
    forall q_0 \in I_a call dfs_blue(q_0);

procedure dfs_blue (State q)
    q.blue := true;
    forall q' \in \text{post}(q) do
        if \neg q'.blue then
            call dfs_blue(q');
        if q \in F_a then
            seed := q;
            call dfs_red(q);

procedure dfs_red (State q)
    q.red := true;
    forall q' \in \text{post}(q) do
        if \neg q'.red then
            call dfs_red(q');
        else if q' = seed then
            report cycle;
```

Diagram:

```
seed
```

Diagram:

```
seed
```
procedure nested_dfs(BA a)
    forall q₀ ∈ Iₐ call dfs_blue(q₀);

procedure dfs_blue (State q)
    q.blue := true;
    forall q' ∈ post(q) do
        if ¬q'.blue then
            call dfs_blue(q');
        if q ∈ Fₐ then
            seed := q;
            call dfs_red(q);

procedure dfs_red (State q)
    q.red := true;
    forall q' ∈ post(q) do
        if ¬q'.red then
            call dfs_red(q');
        else if q' = seed then
            report cycle;

seed
procedure nested_dfs(BA a)
    forall \( q_0 \in I_a \) call dfs_blue\((q_0)\);

procedure dfs_blue (State \( q \))
    \( q.blue := true \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.blue \) then
            call dfs_blue\((q')\);
        else if \( q' = \text{seed} \) then
            report cycle;

procedure dfs_red (State \( q \))
    \( q.red := true \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.red \) then
            call dfs_red\((q')\);
        else if \( q' = \text{seed} \) then
            report cycle;
procedure nested_dfs(BA a)
  forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
  \( q\.blue := true; \)
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'\.blue \) then
      call dfs_blue(\( q' \));
    else if \( q' = \text{seed} \) then
      report cycle;

  if \( q \in F_a \) then
    seed := q;
    call dfs_red(q);

procedure dfs_red (State \( q \))
  \( q\.red := true; \)
  forall \( q' \in \text{post}(q) \) do
    if \( \neg q'\.red \) then
      call dfs_red(\( q' \));
    else if \( q' = \text{seed} \) then
      report cycle;
procedure nested_dfs(BA \ a) 
forall \ q_0 \in I \ a \ call \ dfs\_blue(q_0); 

procedure dfs\_blue (State q) 
q.blue := true; 
forall \ q' \in post(q) \ do 
if \ q'.blue \ then 
call dfs\_blue(q'); 
if \ q \in F \ a \ then 
seed := q; 
call dfs\_red(q); 

procedure dfs\_red (State q) 
q.red := true; 
forall \ q' \in \ post(q) \ do 
if \ q'.red \ then 
call dfs\_red(q'); 
else if \ q' = seed \ then 
report cycle;

\begin{tikzpicture}
\node[shape=circle,draw=black] (A) at (0,0) {};
\node[shape=circle,draw=black] (B) at (2,0) {};
\node[shape=circle,draw=black] (C) at (4,0) {};
\node[shape=circle,draw=black] (D) at (6,0) {};
\node[shape=circle,draw=black] (E) at (8,0) {};
\node[shape=circle,draw=black] (F) at (10,0) {};
\node[shape=circle,draw=black] (G) at (12,0) {};
\node[shape=circle,draw=black] (H) at (14,0) {};

\path[blue,ultra thick,->] (A) edge (B);
\path[blue,ultra thick,->] (B) edge (C);
\path[blue,ultra thick,->] (C) edge (D);
\path[blue,ultra thick,->] (D) edge (E);
\path[blue,ultra thick,->] (E) edge (F);
\path[blue,ultra thick,->] (F) edge (G);
\path[blue,ultra thick,->] (G) edge (H);
\path[red,ultra thick,->] (B) edge (C);
\path[red,ultra thick,->] (C) edge (D);
\path[red,ultra thick,->] (D) edge (E);
\path[red,ultra thick,->] (E) edge (F);
\path[red,ultra thick,->] (F) edge (G);
\path[red,ultra thick,->] (G) edge (H);
\path[black,ultra thick,->] (A) edge (H);
\path[black,ultra thick,->] (B) edge (A);
\path[black,ultra thick,->] (C) edge (B);
\path[black,ultra thick,->] (D) edge (C);
\path[black,ultra thick,->] (E) edge (D);
\path[black,ultra thick,->] (F) edge (E);
\path[black,ultra thick,->] (G) edge (F);
\path[black,ultra thick,->] (H) edge (G);
\node[below] at (6,0) {seed};
\end{tikzpicture}
procedure nested_dfs(BA $a$)
forall $q_0 \in I_a$ call dfs_blue($q_0$);

procedure dfs_blue (State $q$)
$q$.blue := true;
forall $q' \in post(q)$ do
  if $\neg q'.blue$ then
    call dfs_blue($q'$);
  else if $q' = seed$ then
    report cycle;

procedure dfs_red (State $q$)
$q$.red := true;
forall $q' \in post(q)$ do
  if $\neg q'.red$ then
    call dfs_red($q'$);
  else if $q' = seed$ then
    report cycle;
procedure nested_dfs(BA a)
    forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
    \( q\.blue := \text{true} \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.blue \) then
            call dfs_blue(\( q' \));
        if \( q \in F_a \) then
            \( \text{seed := } q \);
            call dfs_red(\( q \));

procedure dfs_red (State \( q \))
    \( q\.red := \text{true} \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.red \) then
            call dfs_red(\( q' \));
        else if \( q' = \text{seed} \) then
            report cycle;

seed
procedure nested_dfs(BA a)
    forall $q_0 \in I_a$ call dfs_blue($q_0$);

procedure dfs_blue (State q)
    $q$.blue := true;
    forall $q' \in post(q)$ do
        if $\neg q'.blue$ then
            call dfs_blue($q'$);
        if $q \in F_a$ then
            seed := q;
            call dfs_red(q);

procedure dfs_red (State q)
    $q$.red := true;
    forall $q' \in post(q)$ do
        if $\neg q'.red$ then
            call dfs_red(q');
        else if $q' = seed$ then
            report cycle;

seed
procedure nested_dfs(BA a) 
    forall $q_0 \in I_a$ call dfs_blue($q_0$); 

procedure dfs_blue (State $q$) 
    $q$.blue := true; 
    forall $q' \in \text{post}(q)$ do 
        if $\neg q'.blue$ then 
            call dfs_blue($q'$); 
        if $q \in F_a$ then 
            seed := $q$; 
            call dfs_red($q$); 

procedure dfs_red (State $q$) 
    $q$.red := true; 
    forall $q' \in \text{post}(q)$ do 
        if $\neg q'.red$ then 
            call dfs_red($q'$); 
        else if $q' = \text{seed}$ then 
            report cycle; 

seed
procedure nested_dfs(BA a)
    forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
    \( q.blue := \text{true}; \)
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.blue \) then
            call dfs_blue(\( q' \));
        if \( q \in F_a \) then
            seed := \( q \);
            call dfs_red(\( q \));

procedure dfs_red (State \( q \))
    \( q.red := \text{true}; \)
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.red \) then
            call dfs_red(\( q' \));
        else if \( q' = \text{seed} \) then
            report cycle;

\( \text{seed} \)
procedure nested_dfs(BA \( a \))
    forall \( q_0 \in I_a \) call dfs_blue\((q_0)\);

procedure dfs_blue (State \( q \))
    \( q\.blue := true; \)
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.blue \) then
            call dfs_blue\((q')\);
        if \( q \in F_a \) then
            seed := \( q \);
            call dfs_red\((q)\);

procedure dfs_red (State \( q \))
    \( q\.red := true; \)
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.red \) then
            call dfs_red\((q')\);
        else if \( q' = \text{seed} \) then
            report cycle;
procedure nested_dfs(BA a)
  forall $q_0 \in I_a$ call dfs_blue($q_0$);

procedure dfs_blue (State $q$)
  $q.blue := true$
  forall $q' \in \text{post}(q)$ do
    if $\neg q'.blue$ then
      call dfs_blue($q'$);
    if $q \in F_a$ then
      $seed := q$;
      call dfs_red($q$);

procedure dfs_red (State $q$)
  $q.red := true$
  forall $q' \in \text{post}(q)$ do
    if $\neg q'.red$ then
      call dfs_red($q'$);
    else if $q' = seed$ then
      report cycle!

$q' = seed$
procedure nested_dfs(BA \( a \))
    forall \( q_0 \in I_a \) call dfs_blue(\( q_0 \));

procedure dfs_blue (State \( q \))
    \( q.blue := true \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.blue \) then
            call dfs_blue(\( q' \));
        if \( q \in F_a \) then
            seed := \( q \);
            call dfs_red(\( q \));

procedure dfs_red (State \( q \))
    \( q.red := true \);
    forall \( q' \in \text{post}(q) \) do
        if \( \neg q'.red \) then
            call dfs_red(\( q' \));
        else if \( q' = \text{seed} \) then
            report cycle;

alternative: report cycle earlier, when dfs_red() encounters state that is on dfs_blue() stack – requires extra data structure for that stack.
SCC-based- vs. Nested DFS Approach

• SCC-based approach:
  – finds shorter accepting runs – why is this good?
  – Good because these are the counter-examples that help us understand how a property is violated

• Nested DFS approach:
  – better suited for “on-the-fly” emptyness checks
    • BA is constructed while exploring it
    • accepting runs may be found before whole BA is explored/constructed
  – Spin uses a modified version of the Nested DFS algorithm

• Further work on efficient emptiness checking
Agenda

1. Introduce Büchi Automata (✓)
2. Checking emptyness of the language accepted by a BA (✓)
3. Product construction for BAs
4. Represent KS as BA
5. Represent LTL formula as BA
Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$

- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$

- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r, q, x), \sigma, (r', q', y)) \in T$ iff
    - $(r, \sigma, r') \in T_1$ and $(q, \sigma, q') \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

• Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$

• Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff

    • $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    • $x = 0$ and $r_m \in F_1$, then $y = 1$
    • $x = 1$ and $q_n \in F_2$, then $y = 2$
    • $x = 2$ then $y = 0$
    • otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$
- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$

- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$
- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$
- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$
- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

Product Construction for BA

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$
- Building an automaton $B_1 \otimes B_2$ that accepts $L(B_1) \cap L(B_2)$:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma, T, I_1 \times I_2 \times \{0\}, Q_1 \times Q_2 \times \{2\})$
  - we have $((r_i, q_j, x), \sigma, (r_m, q_n, y)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$
    - $x = 0$ and $r_m \in F_1$, then $y = 1$
    - $x = 1$ and $q_n \in F_2$, then $y = 2$
    - $x = 2$ then $y = 0$
    - otherwise $x = y$

1. Introduce Büchi Automata (√)
2. Checking emptiness of the language accepted by a BA (√)
3. Product construction for BAs (√)
4. Represent KS as BA
5. Represent LTL formula as BA
Represent a Kripke Structure as a Büchi Automaton

• This is quite simple – an example:

Special Case for BA Product Construction

- Product construction can be simplified if all states of one automaton are accepting
  - In the case all states of the automaton of the modeled system are accepting

- Given two BA $B_1 = (Q_1, \Sigma, T_1, I_1, F_1)$ and $B_2 = (Q_2, \Sigma, T_2, I_2, F_2)$

- If $F_1 = Q_1$, then $B_1 \otimes B_2$ is defined as follows:
  - $B_1 \otimes B_2 = (Q_1 \times Q_2, \Sigma, T, I_1 \times I_2, Q_1 \times F_2)$
  - we have $((r_i, q_j), \sigma, (r_m, q_n)) \in T$ iff
    - $(r_i, \sigma, r_j) \in T_1$ and $(q_m, \sigma, q_n) \in T_2$

Accepting where second automaton is accepting
Both automata “agree” on transition, as usual
1. Introduce Büchi Automata (✓)
2. Checking emptiness of the language accepted by a BA (✓)
3. Product construction for BAs (✓)
4. Represent KS as BA (✓)
5. Represent LTL formula as BA

next time (in two weeks)
ICSE 2013
35th International Conference on Software Engineering

May 18–26, 2013
San Francisco, CA
Hyatt Regency
(in the heart of the Embarcadero District)

http://2013.icse-conferences.org

Plan now to come to ICSE,
watch our web site for updates,
and follow us in the
social media @ICSEconf and

“San Francisco has only one drawback.
‘Tis hard to leave”, Rudyard Kipling

“If you’re alive, you can’t be bored in
San Francisco. If you’re not alive, San Francisco
will bring you to life.”, William Saroyan
Our Paper at SEAMS

change in requirements or environment assumptions
(assumption or requirement MSDs added or removed)

dynamically updating controller

is implemented by

current controller (c)

Specifications

Specification S

Specification S'