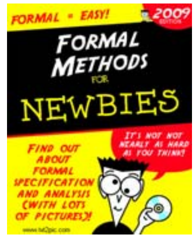
# CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification




## Topic 3: Intro to BIR and Bogor

**Juergen Dingel**
**Jan, 2009**



# Modeling Behaviour of Systems

- Where are we?
  - We've decided to use FSAs to model the behaviour of software systems
  - Have seen:
    - Definition
    - Two types of parallel composition
    - Various related alternatives
- What's next?
  - But, to be able to feed FSAs into a model checker, we need to be able to express FSAs textually in some language
  - Also, it would be nice if that language was as high-level (user-friendly) as possible.
  - 2 examples for modeling languages based on FSAs:
    - BIR (used by Bogor model checker)
    - Promela (used by Spin model checker)



# CISC853: Contents

1. Concurrency
2. Modeling: How to describe behaviour of a software system?
   - finite automata
3. Intro to 2 software model checkers
   - Bogor (Santos group at Kansas State University)
   - Spin (G. Holzmann at JPL)
4. Model checking I
   - algorithms for basic exploration
5. Specifying: How to express properties of a software system?
   - assertions, invariants, safety and liveness properties
   - Linear temporal logic (LTL) and Buechi automata
6. Model checking II
   - algorithms for checking properties
7. Overview of Software Model Checking tools



# BIR, Bogor, and Bandera

- BIR (Bandera Intermediate Representation) is the input language used by the Bogor model checker
- Bogor is the model checker used for the next generation of Bandera
- Bandera is a model checking framework for Java programs
  - automatic translation of
    - Java programs into BIR
    - BIR counter examples back to Java
  - display of counter examples
  - specification manager
  - automatic optimization (abstraction, slicing)
- All developed at Kansas State University

# More BIR, Please!

- BIR is a guarded command language

  when <condition> do <command>
- Support for standard features of oo-languages, e.g.,
  - dynamically created objects and threads
  - exceptions
  - inheritance
  - locks
  - user-defined data types

  ⇒ reduce the semantic gap between OO-programming languages and input language of model checker
- Support for non-determinism
- Next: BIR syntax and semantics



# Example 1: Dining Philosophers

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  active thread Philosopher1() {
    loc loc0:          // take first fork
      when !fork1 do { fork1 := true; }
      goto loc1;

    loc loc1:          // take second fork
      when !fork2 do { fork2 := true; }
      goto loc2;

    loc loc2:          // put second fork
      do { fork2 := false; }
      goto loc3;

    loc loc3:          // put first fork
      do { fork1 := false; }
      goto loc0;
  }

  active thread Philosopher2() {
    loc loc0:          // take second fork
      when !fork2 do { fork2 := true; }
      goto loc1;

    loc loc1:          // take first fork
      when !fork1 do { fork1 := true; }
      goto loc2;

    loc loc2:          // put first fork
      do { fork1 := false; }
      goto loc3;

    loc loc3:          // put second fork
      do { fork2 := false; }
      goto loc0;
  }}
```

variable declaration

[source: CIS842 @ KSU]



# Example 1: Dining Philosophers (Cont'd)

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  active thread Philosopher1() {
    loc loc0:          // take first fork
      when !fork1 do { fork1 := true; }
      goto loc1;

    loc loc1:          // take second fork
      when !fork2 do { fork2 := true; }
      goto loc2;

    loc loc2:          // put second fork
      do { fork2 := false; }
      goto loc3;

    loc loc3:          // put first fork
      do { fork1 := false; }
      goto loc0;
  }

  active thread Philosopher2() {
    loc loc0:          // take second fork
      when !fork2 do { fork2 := true; }
      goto loc1;

    loc loc1:          // take first fork
      when !fork1 do { fork1 := true; }
      goto loc2;

    loc loc2:          // put first fork
      do { fork1 := false; }
      goto loc3;

    loc loc3:          // put second fork
      do { fork2 := false; }
      goto loc0;
  }}
```

thread declarations (active = thread is started automatically)

[source: CIS842 @ KSU]



# BIR: Guarded Transformations (a.k.a., Guarded Commands)

```
active thread Philosopher1() {
  loc loc0:          // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

  loc loc1:          // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

  loc loc2:          // put second fork
    do { fork2 := false; }
    goto loc3;

  loc loc3:          // put first fork
    do { fork1 := false; }
    goto loc0;
}
```

Control location

When condition is true, …

... then execute statement(s) inside do {...} atomically

Trivially true guard

[source: CIS842 @ KSU]

# BIR: Guarded Transformations (a.k.a., Guarded Commands) (Cont'd)

Can have several transformations per location!

Example:

```
...
loc loc0:
        when x < y do {...}
        goto loc1;
        when x > y do {...}
        goto loc2;
        when x==y do {...}
        goto loc3;
loc loc1:
        ...
```

Part of simplified BIR grammar:

```
<program> ::=      "main"? "active"?
                   "thread" <thread-id>
                   "("<params>?")" <local-var-decls>
                   <location>+

<location> ::=     "loc" <loc-id> ":"  <transform>+

<transform> ::=    <guard>? "do"  "{" <action>* "}"
                   <jump>";" | ...

<guard> ::=        "when" <exp>

<action> ::=       <assignment> | ...

<jump> ::=         "goto" <loc-id> |
                   "return" <local-var-id>
```



# BIR: State

A BIR state consists of…

```
system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  active thread Philosopher1() {
    loc loc0: // take first fork
      when !fork1 do { fork1 := true; }
      goto loc1;

    loc loc1: // take second fork
      when !fork2 do { fork2 := true; }
      goto loc2;

    loc loc2: // put second fork
      ...
      goto loc0;
  }

  active thread Philosopher2() {
    loc loc0: // take second fork
      when !fork2 do { fork2 := true; }
      goto loc1;

    loc loc1: // take first fork
      when !fork1 do { fork1 := true; }
      goto loc2;

    loc loc2: // put first fork
      do { fork1 := false; }
      goto loc3;
      ...
  }}
```

... the values of **global variables** and ...

... for each thread, the **current control location** (program counter) and ...

... for each thread, the values of its **local variables** (but none here)

# BIR Types

- **Supported types**
  - basic: boolean, int, long, float, double
  - range types: int(lower, upper), long(lower, upper)
  - enumeration types: enum cards {spades, hearts, clubs, diamonds}
- **User-defined extension types**
  - primitive
  - reference
    - may be generic (similar to, e.g., generic collections in Java 1.5)
      - Set.type<int> theSet = Set.create<int>(1,2,3,5);
- All types in BIR
  - are bounded (finite) (e.g., int: -2147483648 to 2147483647)
  - have a default value (e.g., int, long: 0)

Very important! (from a theoretical standpoint at least)



# BIR: State Notation

Example:

$[pc_1 \mapsto 0,$
$pc_2 \mapsto 1,$
$fork1 \mapsto false,$
$fork2 \mapsto true]$

…pc for Philosopher1 is loc0
…pc for Philosopher2 is loc1
…value of fork1 is 'false'
…value of fork2 is 'true'

Sometimes abbreviated to

[0, 1, false, true]

...if the ordering of variable values is clear from context

# BIR: Transition Notation

```
active thread Philosopher1() {
  ...

  loc loc2:  // put second fork
    do { fork2 := false; }
    goto loc3;

  loc loc3:  // put first fork
    do { fork1 := false; }
    goto loc0;
}
```

From state:

$[pc_1 \mapsto 2, pc_2 \mapsto 0,$ fork1 $\mapsto$ "true", fork2 $\mapsto$ "true"]

system can make transition into state:

$[pc_1 \mapsto 3, pc_2 \mapsto 0,$ fork1 $\mapsto$ "true", fork2 $\mapsto$ "false"]

**Notation:**

$[pc_1 \mapsto 2, pc_2 \mapsto 0,$ fork1 $\mapsto$ "true", fork2 $\mapsto$ "true"]

$\overset{1:2}{\rightarrow}$ $[pc_1 \mapsto 3, pc_2 \mapsto 0,$ fork1 $\mapsto$ "true", fork2 $\mapsto$ "false "]

The thread Philospher1 executes the transition leading out of loc2



# BIR: Execution Trace

An execution trace is a sequence of transitions between states

$$
\begin{array}{ll}
 & [pc_1 \mapsto 0, pc_2 \mapsto 0, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"false"}] \\
\overset{1:0}{\rightarrow} & [pc_1 \mapsto 1, pc_2 \mapsto 0, \text{fork1} \mapsto \text{"true"}, \text{fork2} \mapsto \text{"false"}] \\
\overset{1:1}{\rightarrow} & [pc_1 \mapsto 2, pc_2 \mapsto 0, \text{fork1} \mapsto \text{"true"}, \text{fork2} \mapsto \text{"true"}] \\
\overset{1:2}{\rightarrow} & [pc_1 \mapsto 3, pc_2 \mapsto 0, \text{fork1} \mapsto \text{"true"}, \text{fork2} \mapsto \text{"false"}] \\
\overset{2:0}{\rightarrow} & [pc_1 \mapsto 3, pc_2 \mapsto 1, \text{fork1} \mapsto \text{"true"}, \text{fork2} \mapsto \text{"true"}] \\
\overset{1:3}{\rightarrow} & [pc_1 \mapsto 0, pc_2 \mapsto 1, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"true"}] \\
\overset{2:1}{\rightarrow} & [pc_1 \mapsto 0, pc_2 \mapsto 2, \text{fork1} \mapsto \text{"true"}, \text{fork2} \mapsto \text{"true"}] \\
\overset{2:2}{\rightarrow} & [pc_1 \mapsto 0, pc_2 \mapsto 3, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"true"}] \\
\overset{2:3}{\rightarrow} & [pc_1 \mapsto 0, pc_2 \mapsto 0, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"true"}] \\
\rightarrow & \ldots
\end{array}
$$



# Semantics: FSA Corresponding to BIR Program

- What is the FSA $A_{DP}$ corresponding to the Dining Philosophers BIR program (DP)?
- $A_{DP} = (S, s_0, L, \delta, F)$ where
  - States S:
    - A total of 64 states:
      - 4 locations for each philosopher (loc0 to loc3)
      - 2 values for each fork
      - total: 4*4*2*2 = 64
    - [0, 0, false, false] to [3, 3, true, true]
  - Initial state $s_0$:
    - each state component has a default initial value:
      - for pc of thread t: the textually first location in the declaration of t
      - for boolean variables: false
      - for integer variables: 0
    - $s_0$ = [0, 0, false, false]



# Semantics: FSA Corresponding to BIR Program (Cont'd)

- $A_{DP} = (S, s_0, L, \delta, F)$ where
  - States S = {[0, 0, false, false], ..., [3, 3, true, true]}
  - Initial state $s_0$ = [0, 0, false, false]
  - Labels L = {i:j | i∈ {0, …, numThreads(DP)-1} ∧ j∈ {0, …, maxNumLocsInThread(DP)-1}

    // here, numThreads(DP)=2, maxNumLocsInThread(DP)=4
  - Transitions δ:
    - Each transition leading out of BIR location *loc* in thread *t* has an implicit guard that only allows it to be enabled when *t*'s program counter is at *loc*
    - Have to see which pairs of states s, s' each transition in the BIR code gives rise to
    - For $A_{DP}$, there are 2*(8+8+16+16)=96 transitions in δ; e.g., thread 1 has 8 transitions of the form ([0,$l_2$,false,$f_2$], [1, $l_2$, true, $f_2$]) out of loc. 0
  - Final states F = {s | s is deadlocked}

Bogor calls deadlocked states "invalid end states"

# Transition Examples

```
active thread Philosopher1() {
  loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

  loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

  loc loc2: // put second fork
    do { fork2 := false; }
    goto loc3;

  loc loc3: // put first fork
    do { fork1 := false; }
    goto loc0;
}
```

We have

([1, 0, “true”, “false”], 1:1,
[2, 0, “true”, “true”]) $\in \delta$

and

([1, 2, “false”, “false”], 1:1,
[2, 2, “false”, “true”]) $\in \delta$

and more



# BIR: Enabled & Disabled Transitions

A BIR transformation

**loc** i:
  **when** b **do** {...}
  **goto** j

of thread t is *enabled* in a particular state s if

- i is the current control location of t, and
- b evaluates to true in s.

**Example:**

```
active thread Philosopher1() {
  loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

  loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

  ...
```

This transformation is disabled on each of:

- [1, 1, “true”, “true”]
- [0, 0, “false”, “false”]
- [1, 2, “false”, “true”]

Why?



# Reachable States and State Space

- Not every state is reachable through a sequence of transitions from the initial state
- For instance, the state
  $[pc_1 \mapsto 2, pc_2 \mapsto 0, fork1 \mapsto$ “false”, $fork2 \mapsto$ “false”]
  is unreachable. Why?
- How many states does the DP examples have?
- How many reachable states does the DP example have?



# Non-determinism Revised

- More than one transition may be enabled in a given state
- Sources of non-determinism in BIR programs:
  - intra-thread: more than one transition in one thread enabled
  - inter-thread: one enabled transition in more than one thread
- Example:

```
int x;
thread T1() {
  loc loc0:
    when x>=0 do {...}
    goto loc1;
    when x==0 do {...}
    return;
  ...
}
```

```
thread T2() {
  loc loc0:
    when x==0 do {...}
    ...
  ...
}
```

3 enabled transitions in states with x=0 and $pc_1$=loc0 and $pc_2$=loc0.
Model checking allows you to explore them all!

# Schedules and Executions

- Schedules describe how non-determinism is resolved, that is, which transitions are taken at each state
- A schedule thus determines an execution
- A program has more than one schedule/execution iff it's non-deterministic
- In general, sources of non-determinism are:
  - inputs
    - from user or other applications
    - at beginning of program and during execution
  - thread scheduling policy



# More BIR, Please!

```
system nDiningPhilosophers {
  record Object { }

  record Fork extends Object {
    boolean isHeld;
  }

  const MAX {
    N = 3;
  }

  thread P(Fork f1, Fork f2) {
    loc loc0:
      when !f1.isHeld do {
             f1.isHeld := true;
      }
      goto loc1;
      ...
  }          // end thread Phil
  ...
```

records

arrays

extension

constants

parameters

state right after transform is invisible

```
main thread MAIN() {
  int c;
  Fork[] forks;

  loc loc0:
    when MAX.N > 1 do invisible {
         forks := new Fork[MAX.N];
    }
    goto loc1;
    when MAX.N <= 1 do {}
    return;
  loc loc1:
    when c == 0 do invisible {...}
    goto loc1;
    when c < MAX.N && c != 0 do invisible {
         forks[c] := new Fork;
         start  P(forks[c-1], forks[c]);
         c := c + 1;
    }
    goto loc1;
    when c == MAX.N do invisible {...}
    return;
  }                       // end thread MAIN
}        // end system nDiningPhilosophers
```



# More BIR, Please! (Cont'd)

Functions in BIR

Declaration

```
function random() returns int {
  int i;
  loc loc0:
    do {i := 0;}
    goto loc1;
    do {i := 1;}
    goto loc1;
  loc loc1:
    do {}
    return i;
}  // end function random
```

Function invocation is a transformation, i.e., it's not inside a when ... do {...}

Result of function invocation must be assigned to local variable!

Use

```
thread t() {
  int c;
  loc loc0:
    c := invoke random()
  goto loc0;
  loc loc1:
    ...
}  // end thread t
```



# More BIR, Please! (Cont'd)

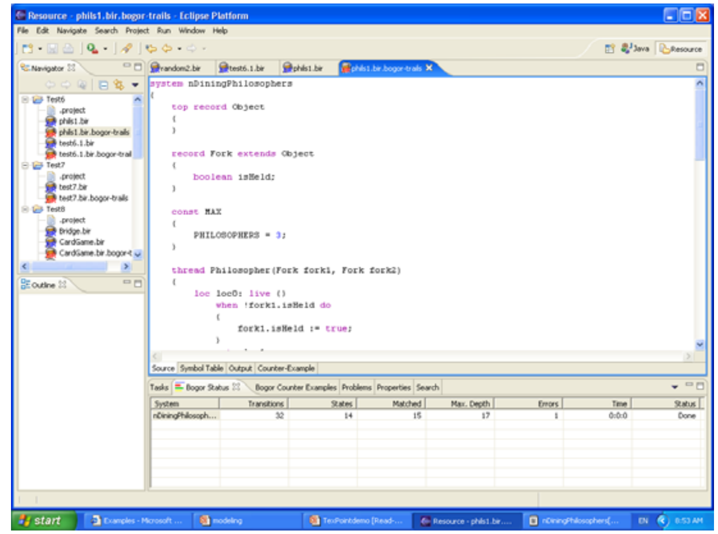
- More info on BIR
  - `http://bogor.projects.cis.ksu.edu`

# Bogor

- Model checker for dynamic and concurrent software
- Developed at Kansas State University
- Features
  - input language directly supports many features of oo-languages, e.g.,
    - dynamic objects and threads, dynamic method dispatch, locking
  - very customizable and modular. Can
    - add new data types: sets, priorities queues, etc
    - change the representation of the state
    - change change the behaviour of the searcher
  - lots of powerful optimizations, e.g.,
    - collapse compression, heap and thread symmetry, partial order reductions
- Already been customized to check
  - Java programs (Bandera project at KSU)
  - real-time avionics systems (Cadena project at KSU)
  - applications using the SIENA publish/subscribe infrastructure (Queen's)



# Bogor (Cont'd)



- Implemented in Java as an Eclipse (IBM) plug-in
- Don't need to know Eclipse (can learn "on the job")

DEMO



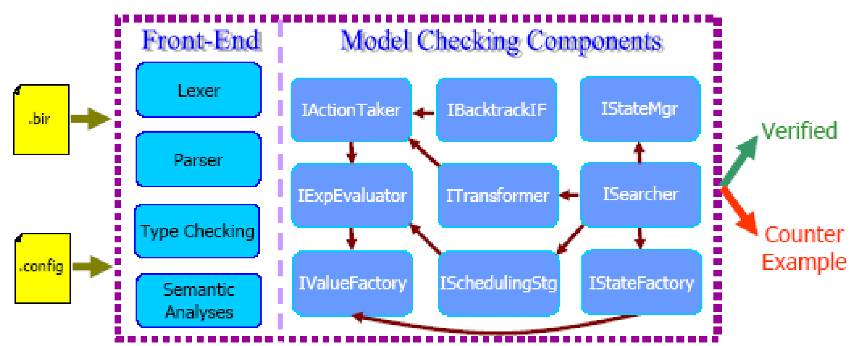
# Bogor (Cont'd)

- Currently, can use Bogor to check for
  - assertion violations
  - invalid endstates (deadlocks)
  - safety properties (more on this later)
  - LTL checking (more on this later)
- Planned for Bogor
  - CTL checking
  - sophisticated counter example display using, e.g., MSCs
  - incorporation into next generation of Bandera (the software model checker for Java)



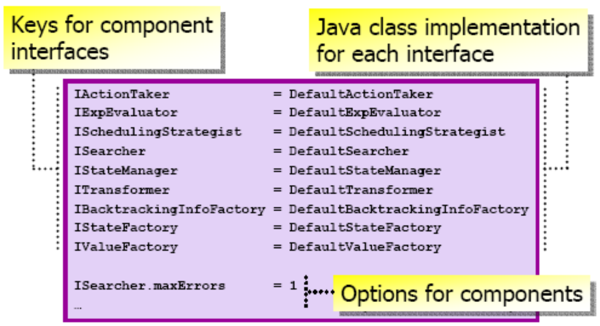
# Bogor Architecture

- **Goal:** modularity and customizability
- Each component has a clearly defined interface



# Configuring Bogor

- A Bogor configuration is a set of pairs (key, value)




- Change configuration by
  - changing the value of a component option
  - providing a different implementation for a component interface



# More Info on BIR and Bogor

- bogor.projects.cis.ksu.edu
  - Bogor software
  - how to install and run Bogor
  - BIR syntax
  - example BIR models

(how to install and run Bogor; BIR syntax: look into Manual)



# In Preparation for Assignment 1

- Go to Bogor website (`bogor.projects.cis.ksu.edu`)
- Download Bogor code
  - accept license agreement
  - create new account
- Install Bogor
  - JRE 1.5, or above
  - Eclipse 3.1, or above
  - GEF 3.0
- Run Bogor on examples provided on Bogor page

(Install Bogor; Run Bogor on examples: • bogor.projects.cis.ksu.edu/manual)



# Forward Reference

- To do Assignment 1, need to know
  - what invariants are and
  - how to check them in Bogor
- Will talk in detail about how to express specifications a bit later
- Next few slides just give you what you need to do Assignment 1

## Types of Formal Specifications for Concurrent and Reactive Systems

- Assertions
- Invariants
  } now (need for A1)
- Safety properties
- Liveness properties
  } later



## Assertions

- Express a property of observables at particular location
- Most basic formal specification; already used by John von Neumann in 1947
- In BIR and Promela: assert(b);
- What kind of correctness claim does an assertion make, that is, what does it mean if there is
  - no assertion violation?:
    *"No matter along which path control has reached the location of the assertion, the boolean expression in the assertion evaluates to true at that location"*
  - an assertion violation?:
    *"There is at least one execution such that the boolean expression in the assertion does not evaluate to true at that location"*

**Example:**

```
thread T() {
  ...
  loc loc7:
    when b do {
      ...
      assert(x>y);
      ...
    }
  ...
}
```



## Example: Checking Mutual Exclusion Using Assertions

- Does protocol below ensure mutual exclusion and deadlock freedom?
- How can we check this using Bogor?

```
system MuxTry {
boolean flag1;
boolean flag2;

thread T1 () {
loc loc0:
do {flag1 := true;} goto loc2;

loc loc2:
when (!flag2) do {} goto loc3;

loc loc3:
do {} goto loc4;

loc loc4:
do {flag1 := false;} goto loc0;
}
```

```
thread T2 () {
loc loc0:
do {flag2 := true;} goto loc2;

loc loc2:
when (!flag1) do {} goto loc3;

loc loc3:
do {} goto loc4;

loc loc4:
do {flag2 := false;} goto loc0;
}
```

*critical regions*



## Example: Checking Mutual Exclusion Using Assertions (Cont'd)

To check mutual exclusion, instrument protocol as follows:

```
system MuxTry {
boolean flag1;
boolean flag2;
int c;

thread T1 () {
loc loc0:
do {flag1 := true;} goto loc2;

loc loc2:
when (!flag2) do {} goto loc3;

loc loc3:
do {c := c+1; assert(c==1);}
goto loc4;

loc loc4:
do {c := c-1; flag1 := false;}
goto loc0;
}
```

```
thread T2 () {
loc loc0:
do {flag2 := true;} goto loc2;

loc loc2:
when (!flag1) do {} goto loc3;

loc loc3:
do {c := c+1; assert(c==1);}
goto loc4;

loc loc4:
do {c := c-1; flag2 := false;}
goto loc0;
}
```

*critical regions*

What about deadlock freedom?

# Detour: Assertions in Java

- Java 1.5 (since 1.4) also supports assertions
- What does it mean if a Java assertion is
  - violated?
  - not violated?
- What's the difference between assertions in Bogor/Spin and Java?



# Invariants

- Express property of observables that holds at every location
- What kind of correctness claim does an invariant make, that is, what does it mean if there is
  - no invariant violation?:

    *"At all locations along all executions of the system, the property holds"*
  - an invariant violation?:

    *"There is at least one location along an execution such that the property does not hold at that location"*
- How do invariants compare to
  - assertions?
  - "loop invariants" in Hoare Logic?



# Multiplication Example

Consider a simple program with a loop invariant

```
// assume parameters m and n
count := m;
output := 0;

// loop invariant: m * n == output + (count * n)
while (count > 0) do {
    output := output + n;
    count := count – 1;
}
```

[Source: CIS842@KSU]



# Multiplication Example

**BIR Version:**

```
system Mult {
int m;
int n;
int count;
int output;

main thread Main () {
loc loc0:
  do {m := (int (0,255)) 5;
      n := (int (0,255)) 4;
      count := m;
      output := (int (0,255)) 0;
      start T1();
  } return;
}
```

Using two threads is unnatural, but the motivation will be clear in a moment…

```
thread T1 () {
loc loc0:
  when (count > 0)
   do {output := output + n;
        count := count - 1;}
  goto loc0;
  when (count == 0) do {}
  return;
}
```

Remember:
No interleaving between these two assignments!

Now, …how to program the check of the invariant?

[Source: CIS842@KSU]

# Checking Invariants

- To check invariant *I* on a program with the threads
  *Main, T1, …, Tn*
  add an assertion of *I* as the last transition of *Main*:

```
main thread Main ()
 ...
 ...
 loc locAssert:
  do {assert (I);}
  return;
```

- Why does this work?
  - Model-checker will explore all possible interleavings between *Main* and each *Ti*
  - Thus, the assertion statement will get interleaved (on some trace) between every pair of execution steps of each *Ti* and thus checking the invariant on every state along every possible execution of *T1, …, Tn*



# Multiplication Example: Checking Invariants

```
system Mult {
 ...
 main thread Main () {
  loc loc0:
   do {m := (int (0,255)) 5;
         n := (int (0,255)) 4;
         count := m;
         output := (int (0,255)) 0;
         start T1();
   }
   goto loc1;

  loc loc1:
    do {assert (m*n ==
                output+(count*n));}
    return;
}
```

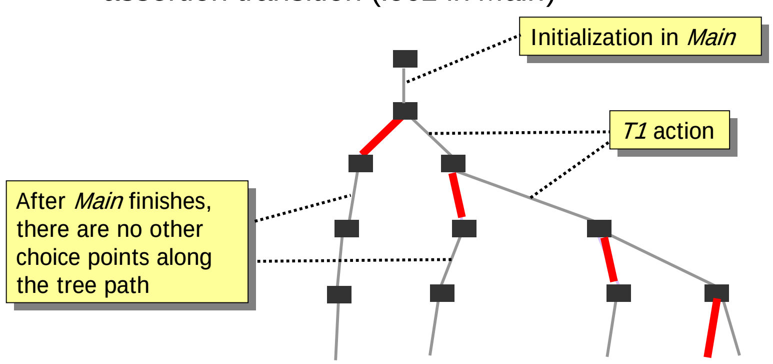
Assertion added

```
thread T1 () {
  loc loc0:
   when (count > 0)  do {
         output := output + n;
         count := count - 1;
   }
   goto loc0;
   when (count == 0)  do {}
   return;
}
```



# Checking Invariants

assertion transition (loc1 in *Main*)




In other words, there exists a path where we do 0 steps of *T1* then check *I*, there exists a path where we do 1 step of *T1* then check *I*, there exists a path where we do 2 steps of *T1*, then check *I*, etc.