CIS 842:
Specification and Verification
of Reactive Systems

Lecture INTRO-DFS:
The Basic Depth-First Search
Algorithm

Copyright 2004, Matt Dwyer, John Hatcliff, and Robby. The syllabus and all lectures for this course are copyrighted
materials and may not be used in other course settings outside of Kansas State University in their current form or modified
form without the express written permission of one of the copyright holders. During this course, students are prohibited
from selling notes to or being paid for taking notes by any person or commercial firm without the express written
permission of one of the copyright holders.

Objectives

- Understand how a system’s state-space can be searched by performing a depth-first search (DFS) on the system’s computation tree.
- Understand the basic structures required for performing a DFS on a computation tree.
- Be able to implement the basic depth-first search algorithm and associated data structures.
- Be able to trace the progress of the depth-first search algorithm on simple BIR-Lite systems.
Outline

- Recalling why we want exhaustive exploration
- Walking through computation trees
- Structure of the state vector
- Structure of the depth-first stack
- Structure of the “seen before” set
- Outline of the basic DFS algorithm

On To Exhaustive Exploration...

- Random simulation
  - isn’t that useful for finding bugs
  - only explores one execution trace
- Guided simulation
  - is only useful on short traces where the user already has a good idea of how a property violation might arise
  - only feasible to explore a few execution traces
- The main strength of model-checking is its automatic exhaustive search capabilities
  - this is why people use model-checkers
  - this is what this course is all about
Exhaustive Depth-first Search

- Bogor can perform exhaustive depth-first searches of a system’s state-space.

At choice points, Bogor chooses an unexplored transition and remembers that it needs to come back and explore the others...

When Bogor has finished with one subtree, ...

... it continues on with the siblings.
Exhaustive Depth-first Search

- Bogor can perform exhaustive depth-first searches of a system’s state-space.

When Bogor has finished with one subtree, ...

... it continues on with the siblings.
Exhaustive Depth-first Search

- Bogor can perform exhaustive depth-first searches of a system’s state-space.

... until the entire computation tree is covered.
DFS with Bogor

For You To Do...

- Pause the lecture...
- Edit `SumToN.bir` and change the assertion to \( x \neq 3 \).
- Use Bogor as described on the previous slide to perform an exhaustive search for property violations on the `SumToN.bir` program.
- What happened? Try to figure out what Bogor’s output is telling you.
  - Did Bogor find an execution path that causes the assertion to be violated?
  - Can you determine what the path is from Bogor’s output?
  - Can you determine how long the path is (how many steps)?
  - What does the other information produced by Bogor tell you?
Bogor Output

Bogor Counterexample Display
Assessment

- Bogor spits out several types information and we haven't covered enough material yet for you to understand what it all means (e.g., the “matched” column).
- We will now discuss the basic data structures associated with the DFS algorithm. This will lead to a good understanding of most of the information that Bogor prints out after a verification run, and will provide a foundation that we will build off of for the rest of this course.

DFS Basic Data Structures

- State vector
  - holds the value of all variables as well as program counters (current position of execution) for each process, and indicates a particular position in the computation tree (as previously covered when discussing state transition systems for BIR-Lite).
- Depth-first stack
  - holds the states (or transitions) encountered down a certain path in the computation tree.
- Seen state set
  - holds the state vectors for all the states that have been checked already (seen) in the depth-first search.

Note: we will represent the values of these data structures in an abstract manner that captures the essence of the issues, but not the actual implementation. Bogor and most other model-checkers actually uses multiple clever representations to obtain a highly space/speed optimized search algorithm.
The state vector is the data structure corresponding to the state (as previously covered when discussing state transition systems for BIR-Lite). It holds the value of all variables as well as program counters for each process, and indicates a particular position in the computation tree.

```
const PARAM { N = 1 };
typealias byte int wrap (0,255);
byte x;
byte t1;
byte t2;

thread Thread1() {
  loc loc0:
  when x != 0 do { t1 := x; } goto loc1;
  loc loc1:
  do { t2 := x; } goto loc2;
  loc loc2:
  do { x := t1 + t2; } goto loc0;
}

thread Thread2() {
  loc loc0:
  when x != 0 do { t1 := x; } goto loc1;
  loc loc1:
  do { t2 := x; } goto loc2;
  loc loc2:
  do { x := t1 + t2; } goto loc0;
}

thread Thread0() {
  loc loc0:
  do { x := 1; } goto loc1;
  loc loc1:
  do { assert (x != PARAM.N); } return;
}
```

Example State Vector: [1,0,2,1,1,0]
SumToN Assertion Violation

Violating schedule for \( N = 2 \)

(initial values)

\[
\begin{align*}
0:0 & \rightarrow [0, 0, 0, x = 0, t1 = 0, t2 = 0] \\
1:0 & \rightarrow [1, 0, 0, x = 1, t1 = 0, t2 = 0] \\
2:0 & \rightarrow [1, 1, 0, x = 1, t1 = 1, t2 = 0] \\
2:1 & \rightarrow [1, 1, 1, x = 1, t1 = 1, t2 = 0] \\
2:2 & \rightarrow [1, 1, 0, x = 2, t1 = 1, t2 = 1] \\
0:1 & \rightarrow [1, 1, 0, x = 2, t1 = 1, t2 = 1]
\end{align*}
\]

...recall state vectors leading to violation of assertion

NOTE: screen shot is for \( N = 1 \), not \( N = 3 \)

Bogor Output

... number of states explored up to when error is found
For You To Do...

- Pause the lecture...
- Give the state vector sequence (as illustrated on the previous slide) for a schedule that leads to a violation of the assertion set to \( \text{assert}(x \neq 3) \).

Depth-first Stack

- The depth-first stack serves two purposes
  - When we come to the end of a path (or a state that we have seen before) and backtrack, the stack tells us where to backtrack to.
  - If an error is encountered, the current value of the stack gives the computation path that leads to the error.
The depth-first stack can be implemented to hold state vectors
- straight-forward implementation

The depth-first stack can be implemented to hold transitions
- requires less space, but ...(see next slide)
Depth-first Stack of Transitions

- Generating a new state requires that the analyzer run a transition on the current state.

- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an “undo” operation to run the transitions in the reverse direction.

- Since the analyzer is not holding states in the stack, when providing variable values as diagnostic information for an error path, the analyzer needs a simulation mode where choice points are decided by the stacked transitions.

Need to revise diagram and states

Depth-first Stack of Transitions

- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an “undo” operation to run the transitions in the reverse direction.
Since the analyzer is not holding states in the stack, when providing variable values as diagnostic information for an error path, the analyzer needs a simulation mode where choice points are decided by the transitions.

Stack of Transitions leading to error state

\[
\begin{align*}
& s_{0.1} = \text{eval}(0.1, s_{\text{init}}) \\
& s_{0.2} = \text{eval}(0.2, s_{0.1}) \\
& s_{0.3} = \text{eval}(0.3, s_{0.2}) \\
& s_{2.1} = \text{eval}(2.1, s_{0.3}) \\
& s_{2.2} = \text{eval}(2.2, s_{2.1}) \\
\end{align*}
\]

Assessment

Many model-checkers (including SPIN and Bogor) implement a depth-first stack of transitions. This reduces amount of required memory and meshes well with its other space optimizations (e.g., bit-state hashing – discussed in following lectures).
Seen State Set

- Often the analyzer will proceed along a different path to a state $S$ that it has checked before.
- In such a case, there is no need to check $S$ again (or any of $S$’s children in the computation tree) since these have been checked before.
- Bogor maintains a **Seen State set** (implemented as a hash table) of states that have been seen before, and it consults this set to avoid exploring/checking a part of the computation tree that is identical to a part that has already been explored before.

Revisting Via A Different Path

```
thread Threadk() {
    loc loc0:
    when x != 0 do { t1 := x; }
    goto loc1;

    loc loc1:
    do { t2 := x; }
    goto loc2;

    loc loc2:
    do { x := t1 + t2; }
    goto loc0;
}

thread Thread0() {
    loc loc0:
    do {
        x := 1;
    }
    goto loc1;

    loc loc1:
    do { assert (x != PARAM.N); }
    return;
}
```

State Vectors in Fragment of Computation Tree

```
[1,0,0,1,0,0] → 1:0
[1,0,1,1,0,0] → 2:0
[1,1,0,1,1,0] → 2:0
[1,1,1,1,1,0] → 1:0
```

...no need to explore this branch because it is identical to one previously explored
Some times we view the computation tree as a graph.

```
thread Threadk() {
    loc loc0:
        when x != 0 do { t1 := x; } goto loc1;

    loc loc1:
        do { t2 := x; } goto loc2;

    loc loc2:
        do { x := t1 + t2; } goto loc0;
    }

thread Thread0() {
    loc loc0:
        do { x := 1; } goto loc1;
    }

    loc loc1:
        do { assert (x != PARAM.N); } return;
}
```

...sharing a node corresponds to (re)visiting a node that has been seen before.

---

**Seen State Set**

```
thread Threadk() {
    loc loc0:
        when x != 0 do { t1 := x; } goto loc1;

    loc loc1:
        do { t2 := x; } goto loc2;

    loc loc2:
        do { x := t1 + t2; } goto loc0;
    }

thread Thread0() {
    loc loc0:
        do { x := 1; } goto loc1;
    }

    loc loc1:
        do { assert (x != PARAM.N); } return;
}
```

...when Bogor gets to this state, it checks the Seen Set and finds it already has been checked, so it backtracks from this point.
Non-Terminating Systems

- Due to the use of the Seen Set, checking a non-terminating system may terminate if the system only has a finite number of states.
- In BIR-Lite, all systems are “finite” because of the bounds on basic data types.
- However, some systems are “more finite” than others.
  - i.e., they have a much smaller state-space.

Consider this example system...

- How many states does it have?
- Does execution of the system terminate?
- Does an exhaustive analysis of the state-space of the system terminate?
For You To Do...

- Pause the lecture...
- Download the file `loops.bir` from the examples page.
- Run Bogor in random simulation mode on the example.
  - What do you observe?
- Run Bogor in model-checking mode.
  - What do you observe?
- Use the output of Bogor to answer the following questions...
  - How many states does the system have?
  - How many states were stored in the Seen Set?
  - How many states does the program generate before it comes back to a previous state?

Bogor Output

...states stored in Seen Set
...generated states that were found to be already in the Seen Set
... # transitions taken during analysis equals # stored states + # generated states seen before
Assessment

- By now you should understand the role of each of DFS basic data structures...
  - State vector
  - Depth-first stack
  - Seen state set
- You should be able to understand what almost all of Bogor’s output means.
- We’ll now re-enforce your intuition behind the main data structures by presenting the pseudo-code for the core of the DFS algorithm as implemented in Bogor.

Core DFS Algorithm

```plaintext
1. seen := \{s_0\}   ...put initial state in seen set
2. pushStack(s_0)    ...current path being explored in computation tree begins with initial state
3. DFS(s_0)          ...

DFS(s)
4. workSet(s) := enabled(s)   ...get the transitions to explore at this state
5. while workSet(s) is not empty
6.     let \( \alpha \in workSet(s) \)
7.     workSet(s) := workSet(s) \- \{\alpha\}
8.     s' := \( \alpha(s) \)   ...calculate the successor state
9.     if s' \( \notin \) seen then
10.    seen := seen \( \cup \) \{s'\}   ...if s’ has not been seen before, then put it in the seen set
11.    pushStack(s')   ...
12.    DFS(s')        ...
13.    popStack()     ...
end DFS              ...
```

...remove s’ from current path
Assessment

- The algorithm on the previous slide can be presented in a number of different ways (recursively, iteratively, some variables as global/local or passed as parameters, etc.), so don’t get caught up in the specific presentation on the previous page.
  - e.g., the value workset(s) must be remembered between different invocations of DFS().
- The presentation on the previous page comes from the book of Clarke et. al.

For You To Do...

- Trace the DFS algorithm on the loops.bir example. Show the value of the three main data structures at each invocation of the DFS method.
Summary

- Bogor is a powerful tool for exploring the state-space of concurrent systems
  - simulation mode
  - verification mode

- Bogor’s three main data structures
  - state vector
    - holds values of variables and program counter for each thread
  - depth-first stack
    - holds states (or transitions) encountered during search
    - used to display the error trace
  - seen set
    - holds states already explored

- Bogor can be used to check for assertion violations, and much more (deadlocks, cycles, temporal logic, etc.)