# Domain-specific Model Checking with Bogor

*SAnToS Laboratory, Kansas State University, USA*

http://bogor.projects.cis.ksu.edu

Robby    Matthew B. Dwyer    John Hatcliff

Matthew Hoosier

Session III: Bogor Architecture

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Department of Defense
Advanced Research Projects Agency (DARPA)

Boeing
Honeywell Technology Center
IBM
Intel

Lockheed Martin
NASA Langley
Rockwell-Collins ATC
Sun Microsystems

# Outline

## Overview

- DFS algorithm
- Bogor components
- Configuration
- Initialization

## State Representation

- Types & Values
- Value Factory
- State Factory

## DFS Stack

- Search algorithm
- Scheduler

## Seen Before Set

- State Manager

# Core DFS Algorithm

$seen := \{s_0\}$ — ...put initial state in seen set

$stack := [s_0]$ — ...current path being explored in computation tree begins with initial state

$DFS(s_0)$

$DFS(s)$

**for each** $\alpha \in enabled(s)$ **do** — ...iterate over all enabled transitions

$s' := \alpha(s)$ — ...calculate the successor state

**if** $s' \notin seen$ **then**

$seen := seen \cup \{s'\}$ — ...if $s'$ has not been seen before, then put it in the seen set

$push(stack, s')$ — ...put $s'$ on the stack which represents the current path in the tree

$DFS(s')$ — ...explore states reachable from $s'$

$pop(stack)$ — ...remove $s'$ from the stack

**end** $DFS$

# Internal Architecture

Front-End

Interpretative Components

Model Checking Components

Lexer

IValueFactory

ISearcher

IStateMgr

.bi

Customizable Checking Engine Modules

Well-formed-ness Checker

IExpEvaluator

ICounterExWr

IProgressMgr

.config

Counter Example

AST Transformer

ITransformer

IEventProvider

IClassLoader

Data-flow Framework

IActionTaker

Extension … Extension

*…modular components with clean and well-designed API using design patterns*

# Bogor Configuration

- A Bogor configuration is a key-value set

Keys for component interfaces

Java class implementation for each interface

```
IActionTaker                = DefaultActionTaker
IExpEvaluator               = DefaultExpEvaluator
ISchedulingStrategist       = DefaultSchedulingStrategist
ISearcher                   = DefaultSearcher
IStateManager               = DefaultStateManager
ITransformer                = DefaultTransformer
IBacktrackingInfoFactory    = DefaultBacktrackingInfoFactory
IStateFactory               = DefaultStateFactory
IValueFactory               = DefaultValueFactory

ISearcher.maxErrors         = 1
…
```

Options for components

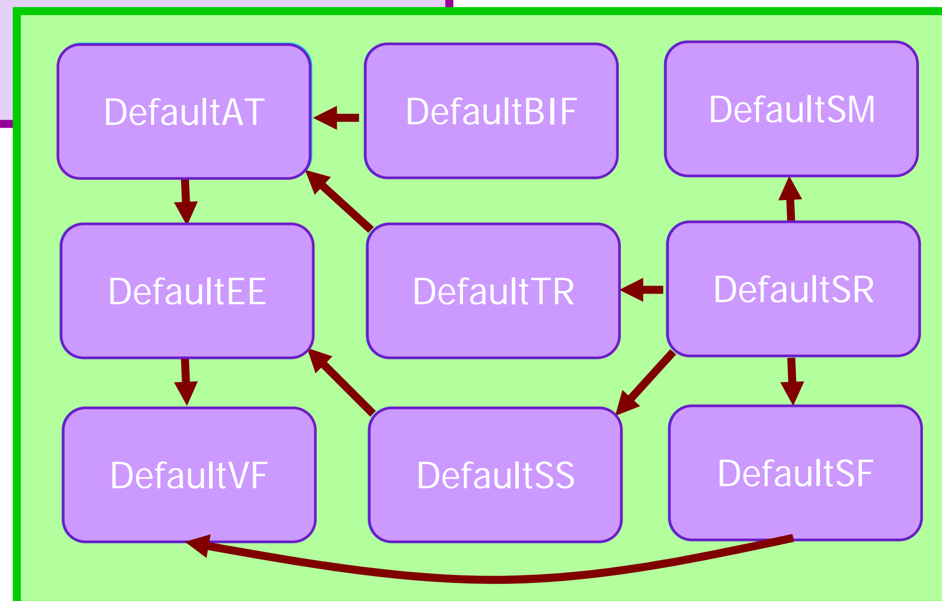
# Bogor Initialization

```
IActionTaker               = DefaultActionTaker
IExpEvaluator              = DefaultExpEvaluator
ISchedulingStrategist      = DefaultSchedulingStrategist
ISearcher                  = DefaultSearcher
IStateManager              = DefaultStateManager
ITransformer               = DefaultTransformer
IBacktrackingInfoFactory = DefaultBacktrackingInfoFactory
IStateFactory              = DefaultStateFactory
IValueFactory              = DefaultValueFactory

ISearcher.maxErrors        = 1
…
```

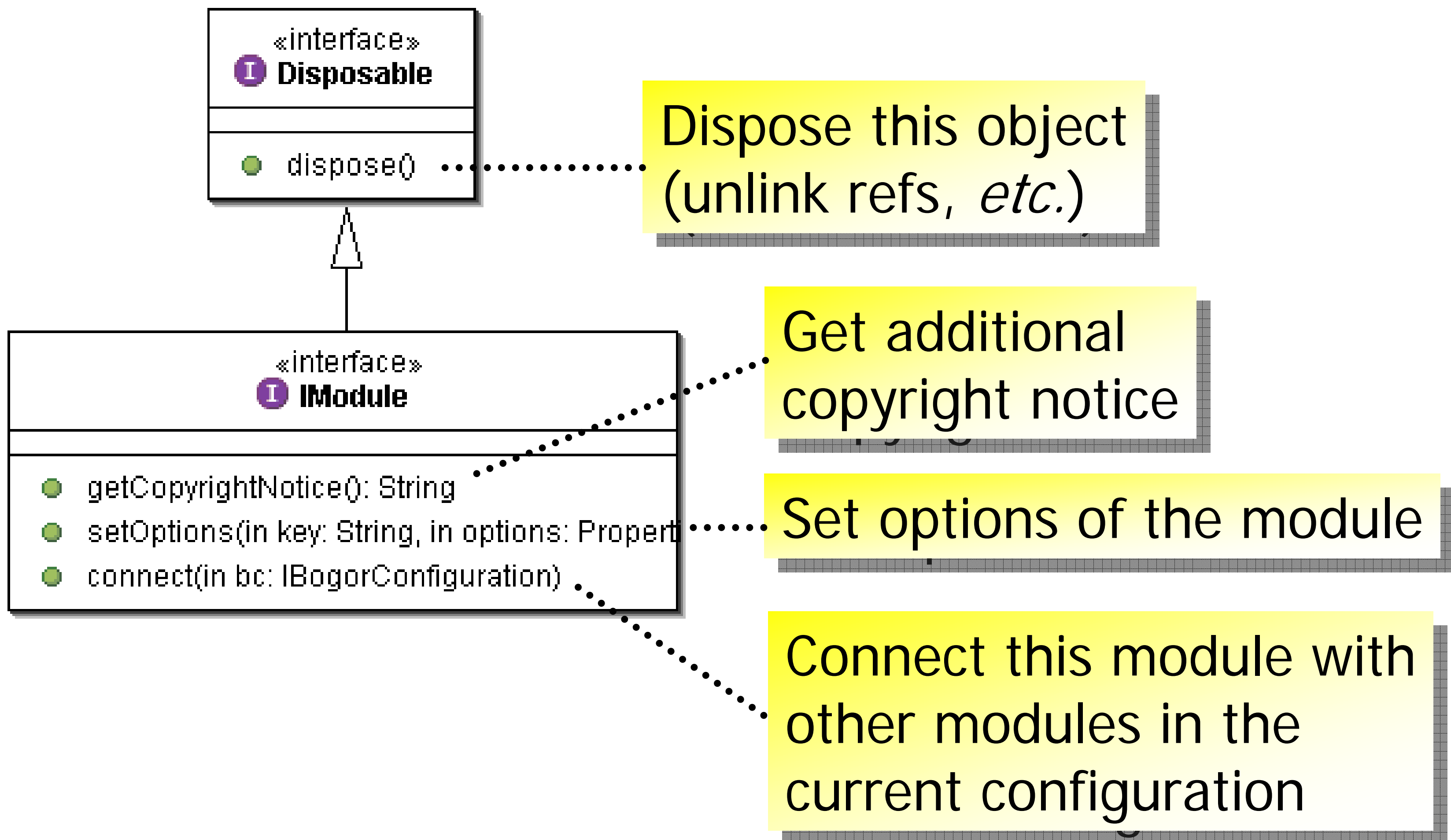
Given a configuration, Bogor instantiate the specified components

Options are passed to each component, and connections are established



# Bogor Module Interface

Each Bogor component must implement IModule



# Outline

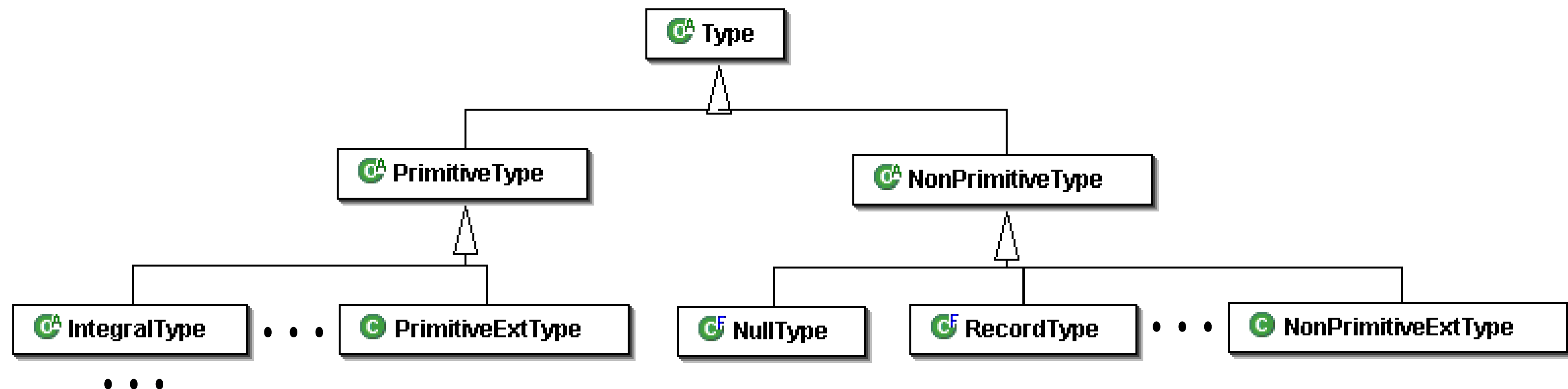
## Overview
- DFS primer
- Bogor components
- Configuration
- Initialization

## State Representation
- Types & Values
- Value Factory
- State Factory

## DFS Stack
- Search algorithm
- Scheduler

## Seen Before Set
- State Manager

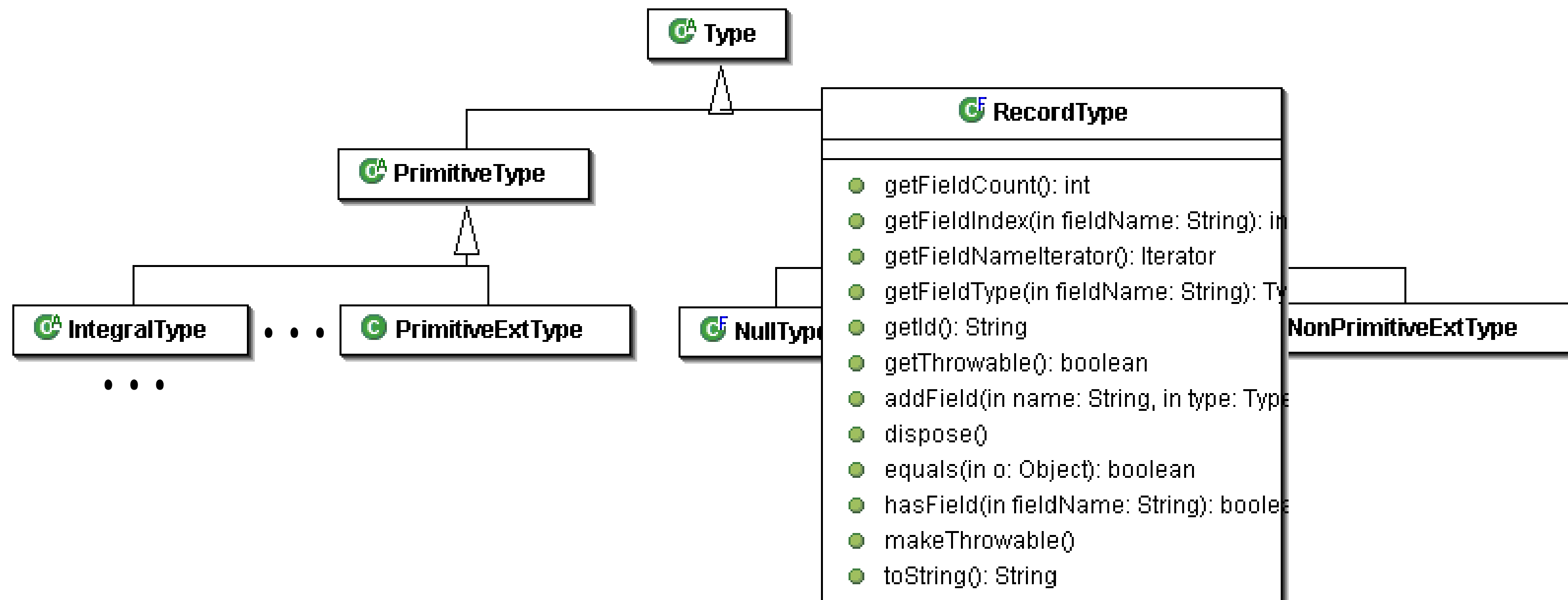
# Type and Value Representations




- BIR types are categorized into two
  - Primitive types: int, long, float, double, *etc.*
  - Non-primitive types: null, record, lock, *etc.*
- Types are created using a TypeFactory
- Each type class has methods to access information about the type being represented

*Package:* **`bogor.type`**

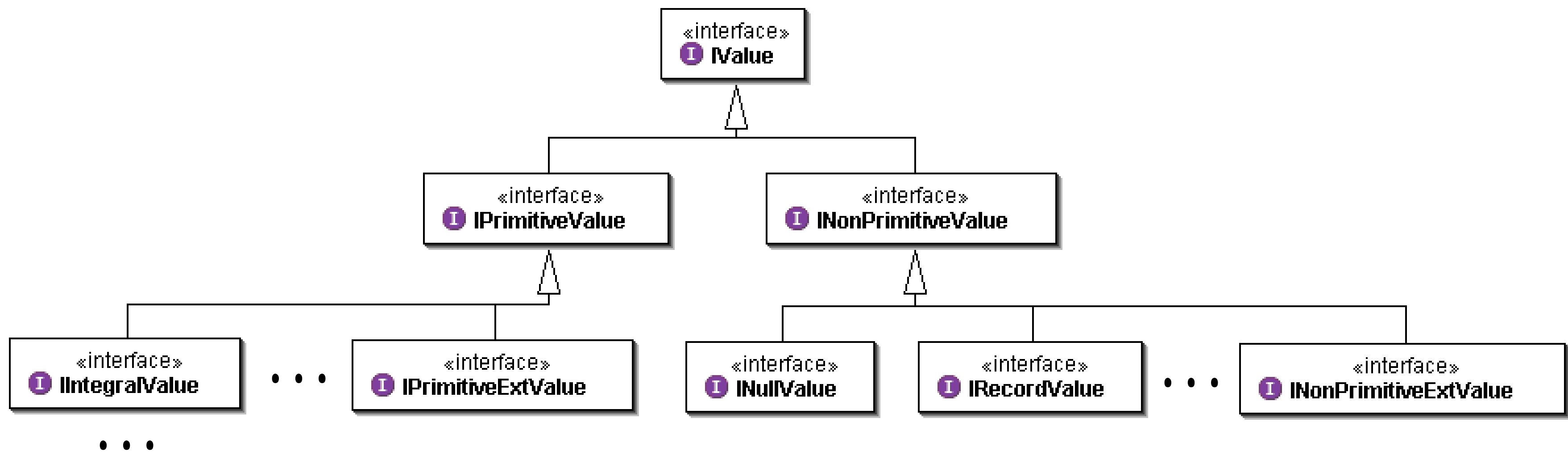
# Type and Value Representations




For example, the record type class has methods to access its fields' names, types, and indices

*Package:* **`bogor.type`**

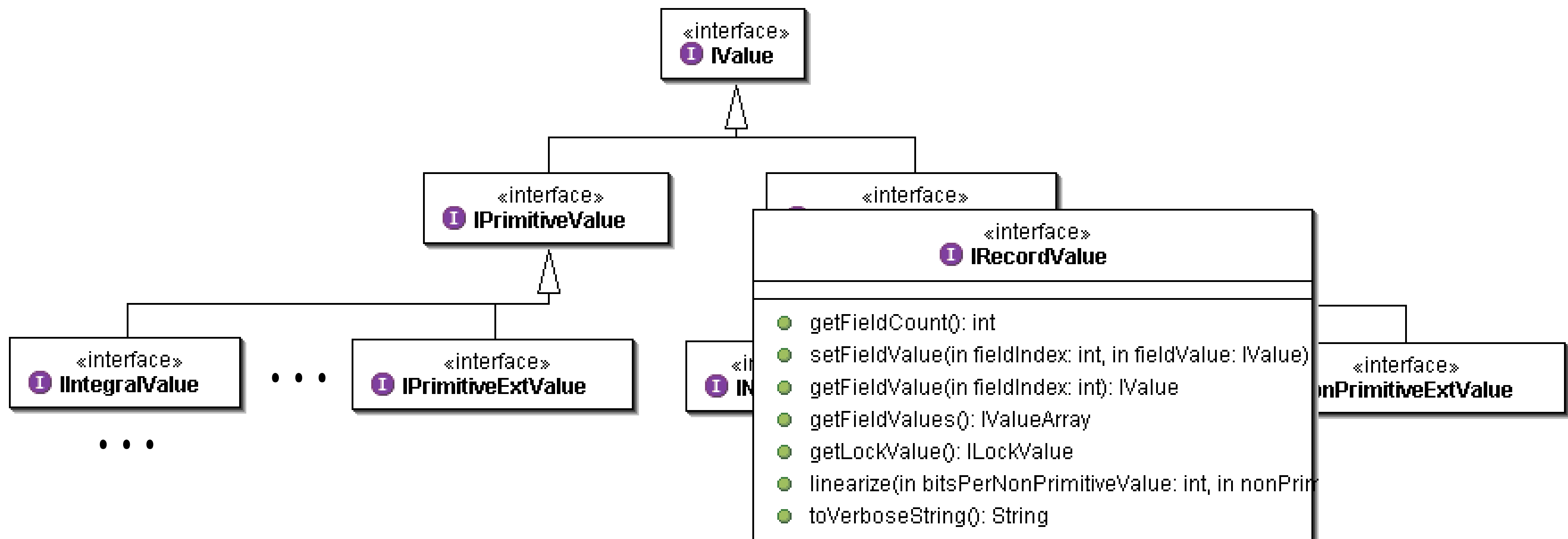
# Type and Value Representations




- BIR values mimic BIR types structure
  - Primitive values: int, long, float, double, *etc.*
  - Non-primitive values: null, record, lock, *etc.*
- Values are created using a ValueFactory

*Package:* **`bogor.module.value`**

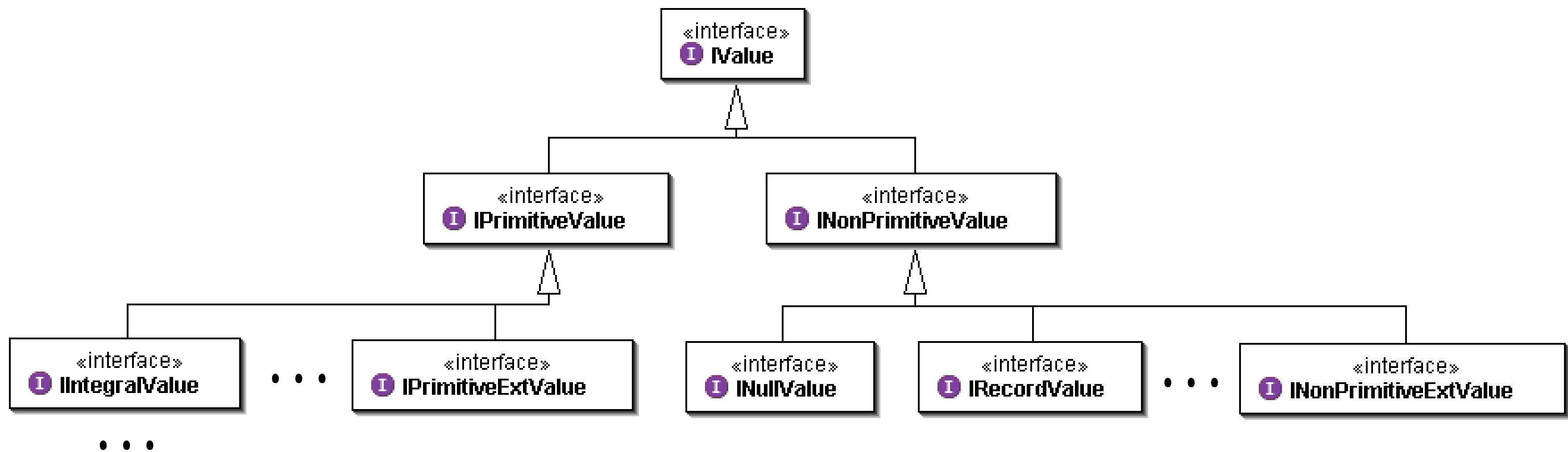
# Type and Value Representations




Similar to the record type, the record value interface has methods to access its fields's values

*Package:* **`bogor.module.value`**

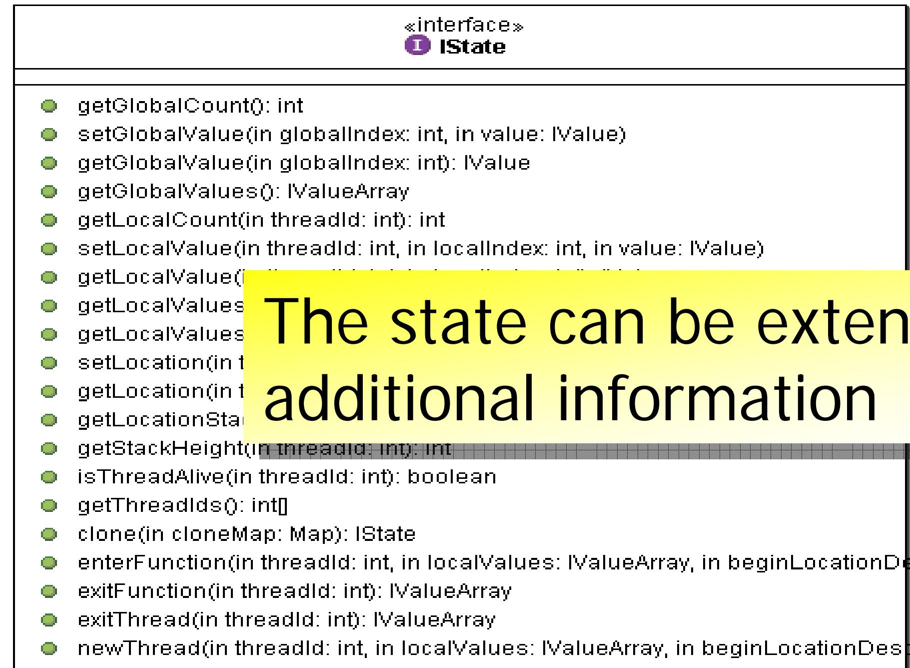
# Type and Value Representations




The hierarchy can be extended to represent, for example, abstract or symbolic values

*Package:* **`bogor.module.value`**

# State Representation




The state can be extended to include additional information

- The state interface has methods to access
  - global values
  - active threads, their program counters and local vars.
  - create or kill threads, and enter or exit functions
- States are created using a StateFactory

*Package:* **`bogor.module.state`**

# Outline

## State Representation

- Types & Values
- Value Factory
- State Factory

## Overview

- DFS algorithm
- Bogor components
- Configuration
- Initialization

## DFS Stack

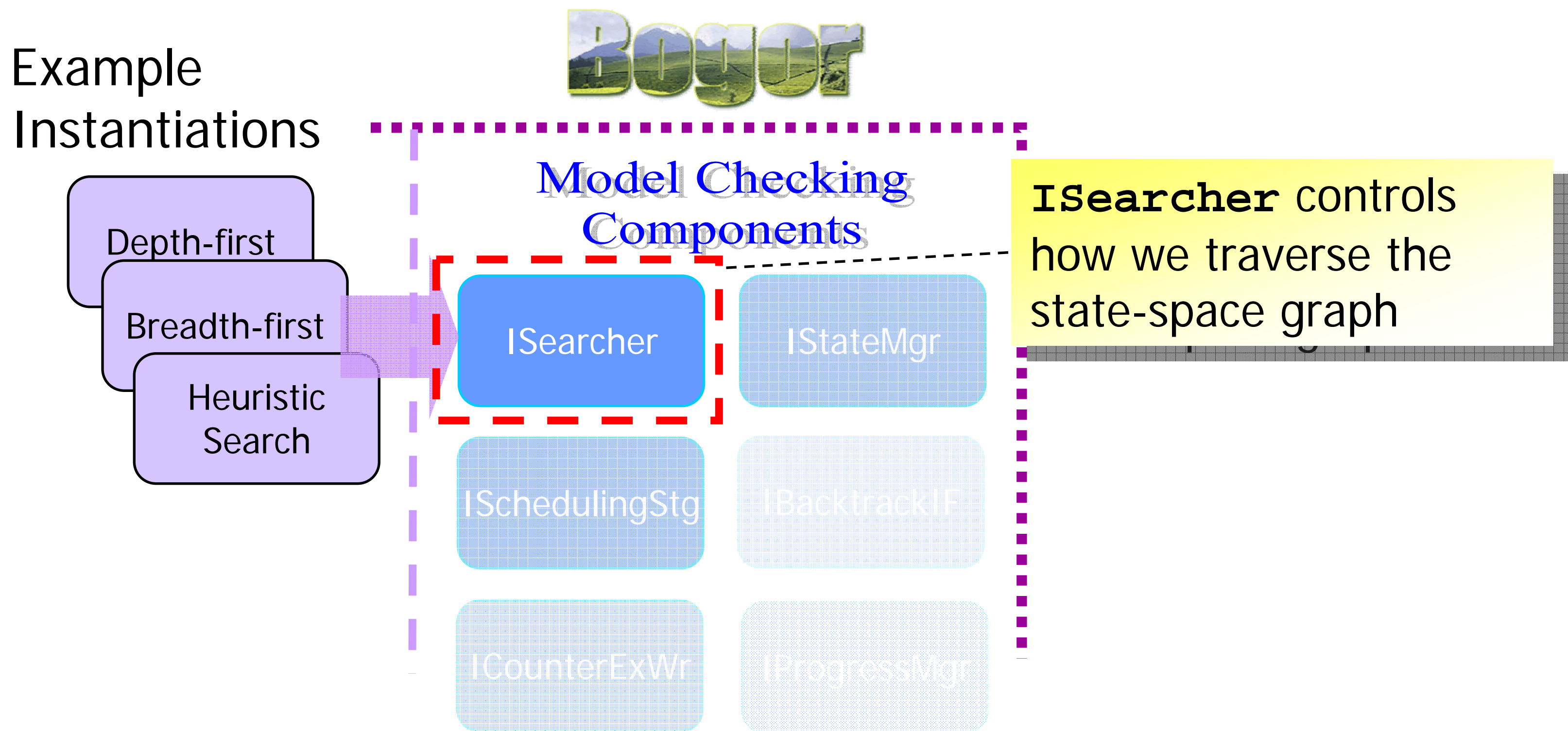
- Search algorithm
- Scheduler

## Seen Before Set

- State Manager

# Navigating The State-space

## The `ISearcher` Module




...customizable checking engine modules

# SimpleSearcher

## A DFS implementation of the state-space exploration

*seen* := $\{s_0\}$
*stack* := $[s_0]$
*DFS* $(s_0)$

*DFS* $(s)$
  **for each** $\alpha \in$ *enabled* $(s)$ **do**
    $s' := \alpha(s)$
    **if** $s' \notin$ *seen* **then**
      *seen* := *seen* $\cup \{s'\}$
      *push* (*stack*, $s'$)
      *DFS* $(s')$
      *pop* (*stack*)
**end** *DFS*

*...abstract version*

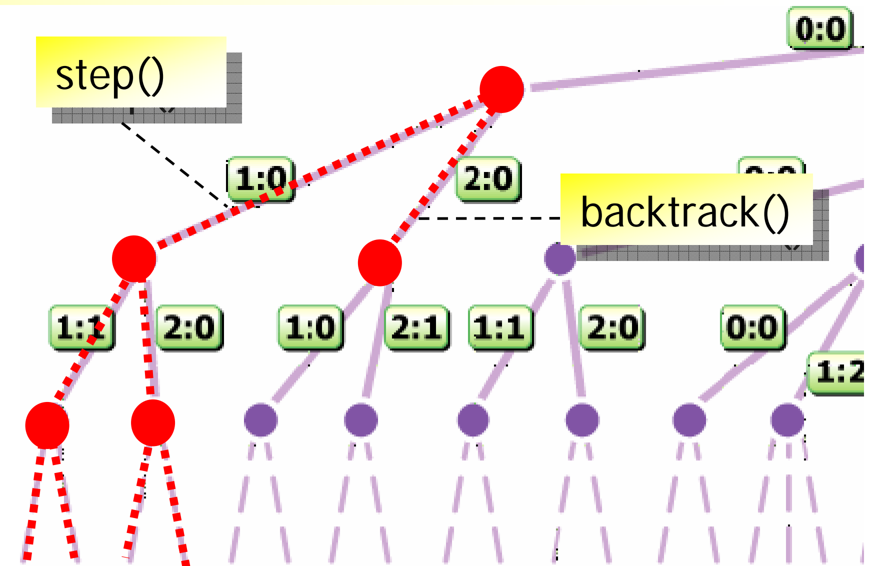
```java
public class SimpleSearcher
       extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  IStateManager sm;
  void initialize() {
    s=createInitialState();
    sm.storeState(s);
  }
  void search() {
    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
```

*...outline of Bogor implementation*

# SimpleSearcher

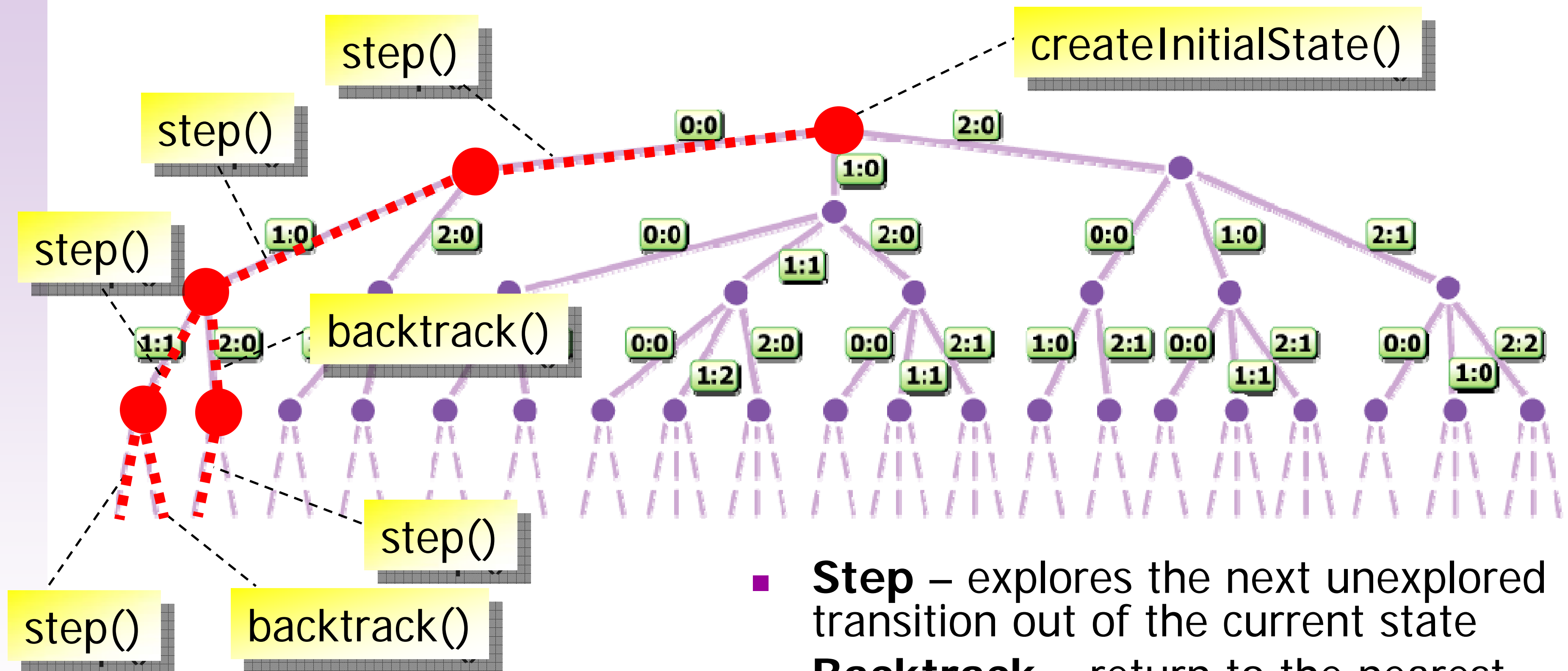
## A DFS implementation of the state-space exploration

```
public class SimpleSearcher
        extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  IStateManager sm;
  void initialize() {
    s=createInitialState();
    sm.storeState(s);
  }
  void search() {
    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
}
```




- **Step** – explores the next unexplored transition out of the current state
- **Backtrack** – return to the nearest state that has not been completely explored, and generate the next unexplored state that descends from the current state

# SimpleSearcher




- **Step** – explores the next unexplored transition out of the current state
- **Backtrack** – return to the nearest state that has not been completely explored, and explore the next unexplored transition out of the state

# Search Module

| ISearcher |
| --- |
| ILL_FORMED_MODEL_CODE: int |
| INVALID_END_STATE_CODE: int |
| ASSERTION_FAILED_CODE: int |
| UNCAUGHT_EXCEPTION_CODE: int |
| RANGE_EXCEPTION_CODE: int |
| EXT_FAILED_CODE: int |
| INVARIANT_VIOLATED_CODE: int |
| getBacktrackingInfos(): ArrayList |
| getErrorCount(): int |
| getState(): IState |
| backtrack(): boolean |
| createInitialState(): IState |
| doTransition(threadDesc: int, t: Transformation, a: Action) |
| error(errCode: int) |
| initialize() |
| search() |
| shouldStore(): boolean |
| step(): boolean |
| store(): boolean |
| writeCounterExamples() |

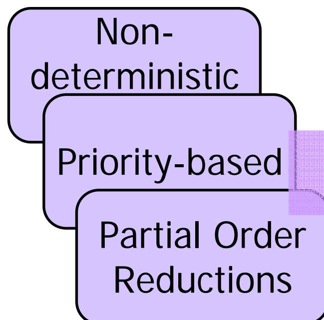
- **`ISearcher`**
  - create initial state
  - step
  - backtrack
- **`SimpleSearcher`**
  - depth-first search
  - iteration-based

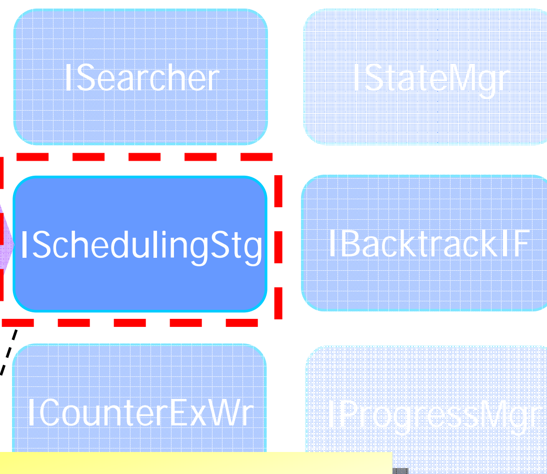
*Package:* **`bogor.module`**

# Scheduling Available Transitions

## The `ISchedulingStrategist` Module

Example Instantiations



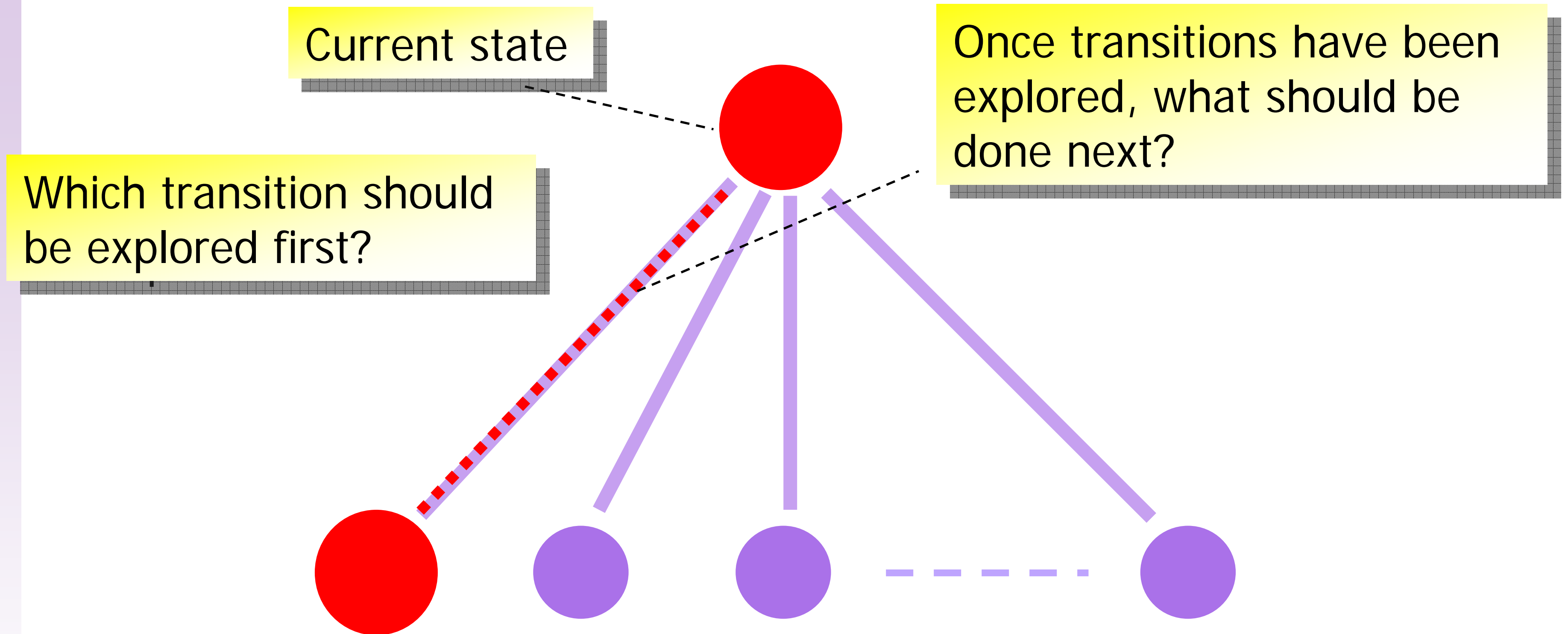

**`ISchedulingStrategist`** picks the next transition to explore at the current state

*seen* := $\{s_0\}$
*stack* := $[s_0]$
*DFS* $(s_0)$

*DFS* $(s)$
  **for each** $\alpha \in$ *enabled* $(s)$ **do**
    $s' := \alpha(s)$
    **if** $s' \notin$ *seen* **then**
      *seen* := *seen* $\cup \{s'\}$
      *push* (*stack*, $s'$)
      *DFS* $(s')$
      *pop* (*stack*)
**end** *DFS*

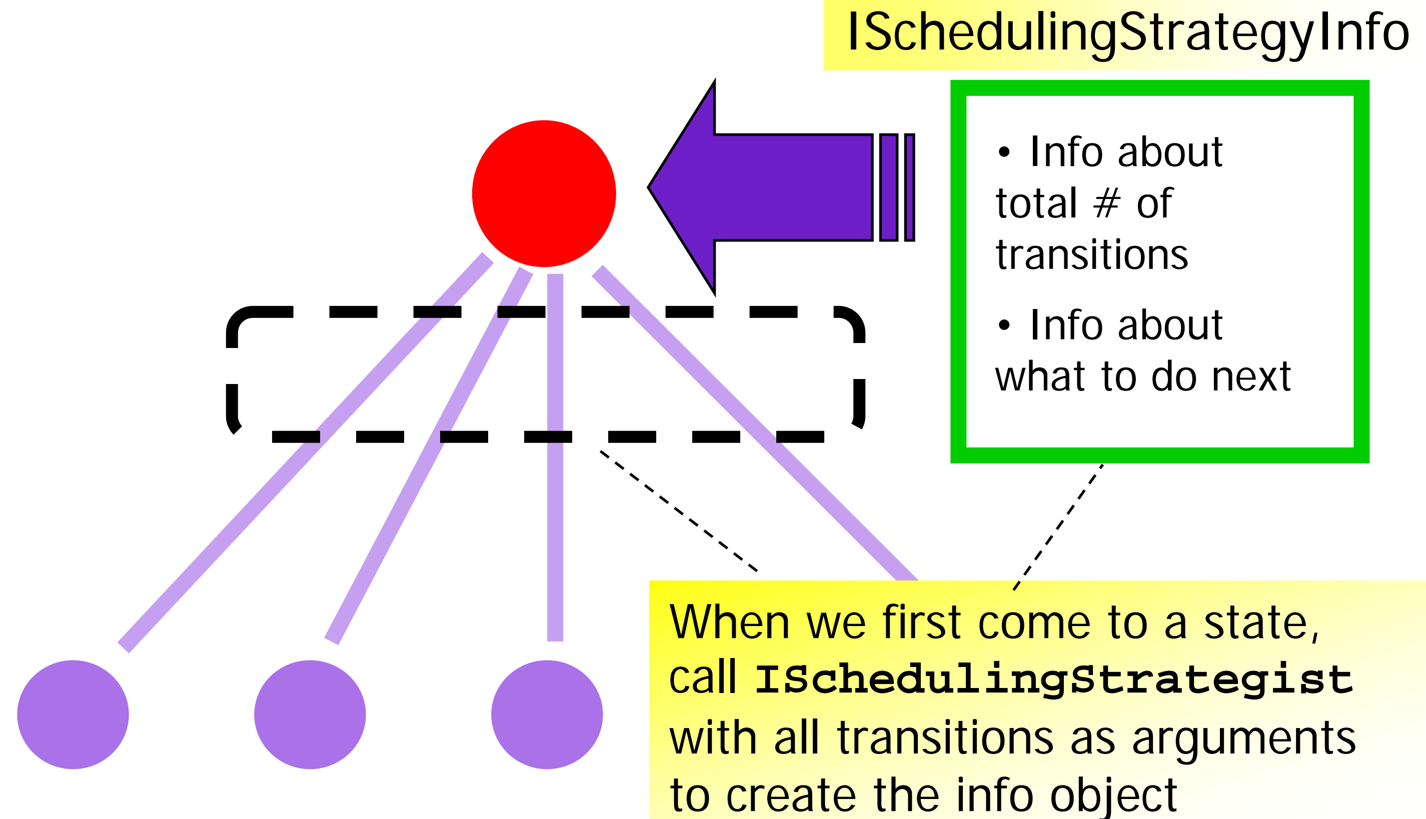
...customizable checking engine modules

# Scheduling Available Transitions




Goal: encapsulate these information and decisions from the rest of the code

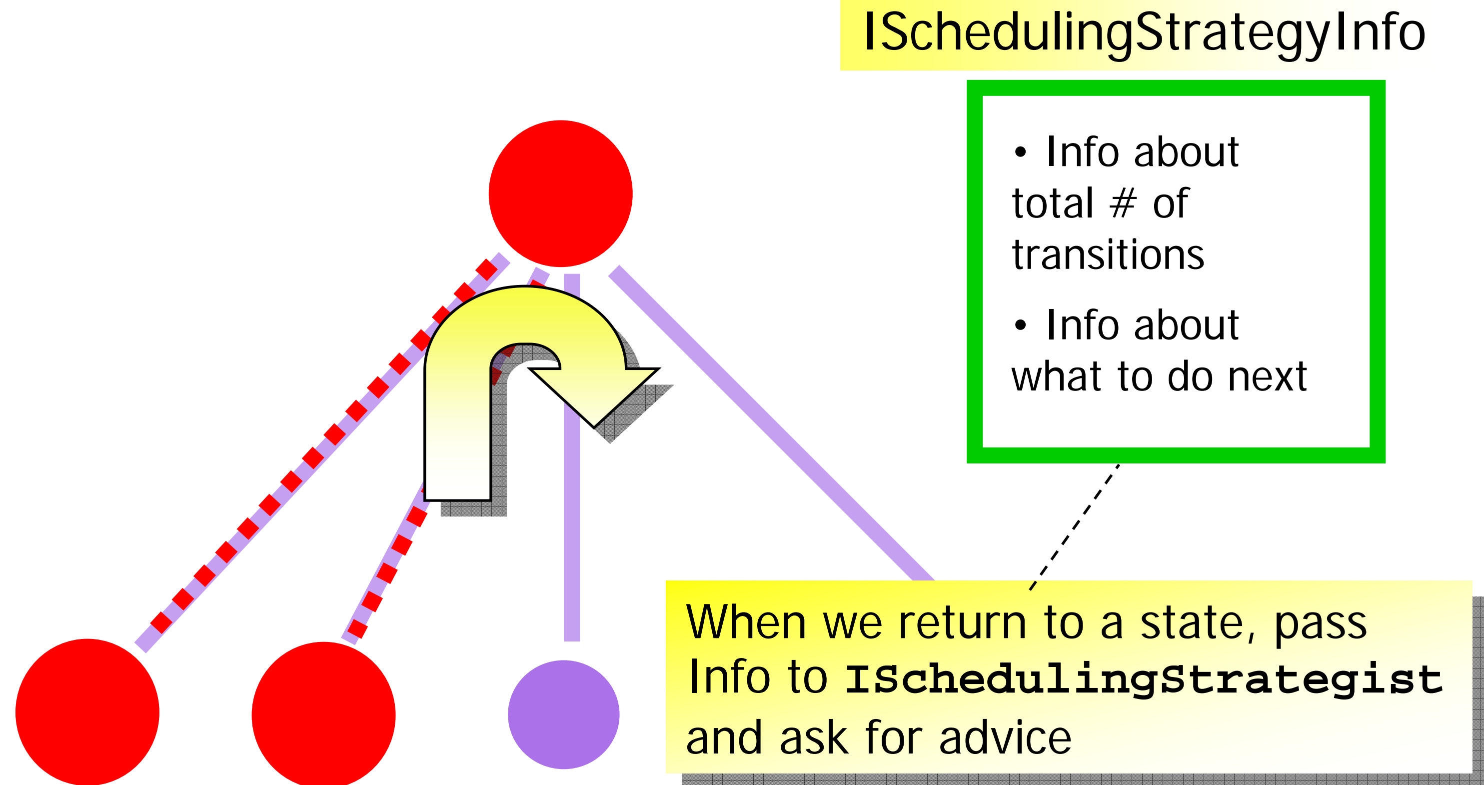
# Scheduling Available Transitions




Encapsulate using an **ISchedulingStrategyInfo** as a Memento

- An object that holds what child transitions have been explored so far
- When **ISearcher** needs to know what to do, call **ISchedulerStrategist** with info and ask for **advice** about what transition to do next

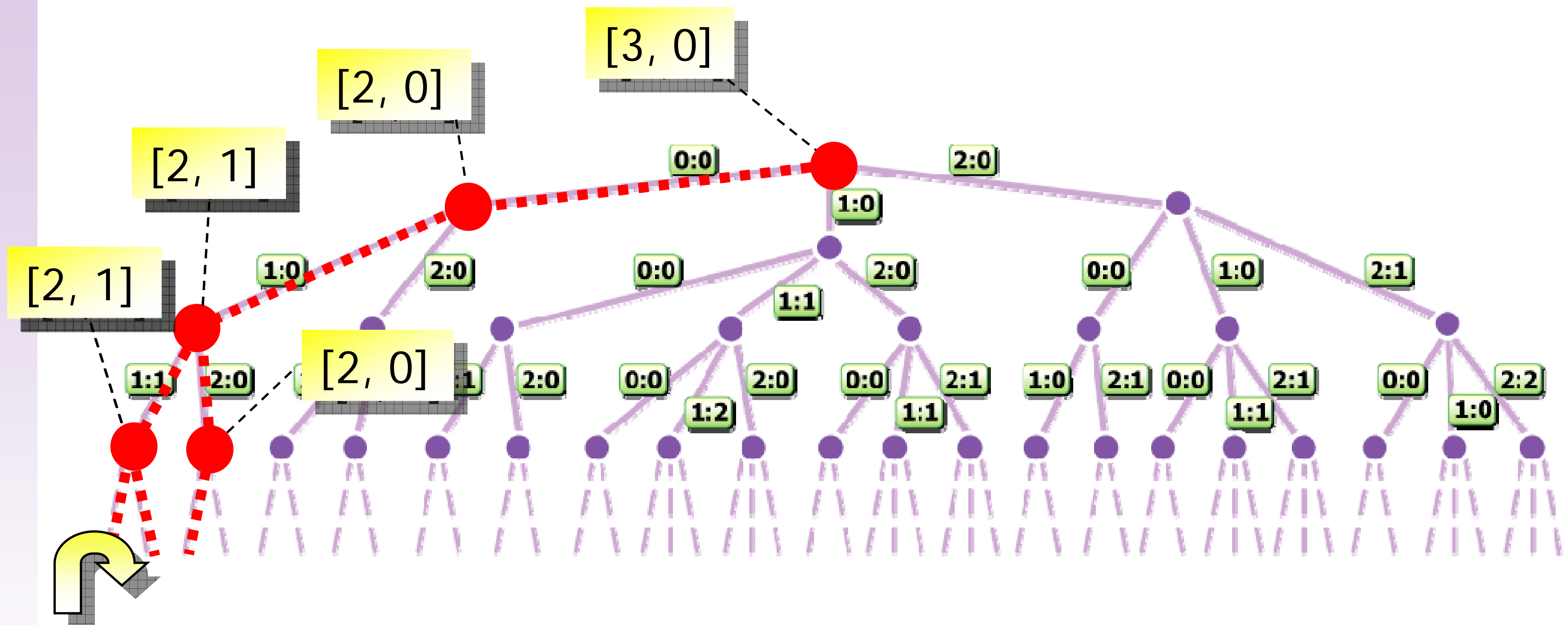
# Scheduling Available Transitions




Encapsulate using an **ISchedulingStrategyInfo** as a Memento

- An object that holds what child transitions have been explored so far
- When **ISearcher** needs to know what to do, call **ISchedulerStrategist** with info and ask for **advice** about what transition to do next

# DefaultSchedulingStrategist & DefaultSchedulingStrategyInfo




Info: [number of enabled transitions, last transition index chosen]

# ISchedulingStrategyInfo

| ISchedulingStrategyInfo |
| --- |
| |
| getChoiceCount(int): int<br>getChosenIndex(int): int<br>isCovered(): boolean<br>getExtId(int): int<br>getInfoCount(): int<br>hasInfo(): boolean<br>clone(Map): ISchedulingStrategyInfo<br>toString(): String |

- Used to keep track
  - whether there is a non-deterministic choice
  - if yes, which transition has been taken

# ISchedulingStrategist

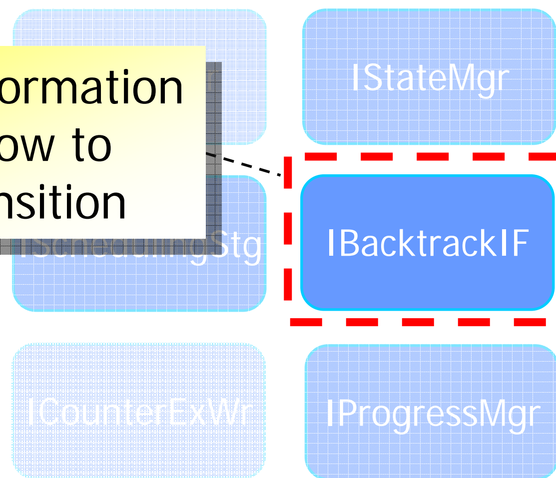
**ISchedulingStrategist** (interface)

- isEnabled(state: IState, t: Transformation, threadId: int): boolean
- getEnabledTransformations(etc: IEnabledTransformationsContext): IntObjectTable
- advise(ssc: ISchedulingStrategyContext, transformations: Transformation[], ssi: ISchedulingStrategyInfo): int
- newStrategyInfo(): ISchedulingStrategyInfo

- Used to:
  - determine enabled transitions
  - determine which transition to take
  - create strategy info
- **DefaultSchedulingStrategist**
  - Full state-space exploration
    - the scheduling policy ensure that each state is visited
  - At each choice point, the info contains
    - the number of enabled transitions
    - the last chosen transition index
  - **advise()** simply increase the last chosen transition index until all are chosen

# Backtracking Information

## The `IBacktrackingInfoFactory` Module



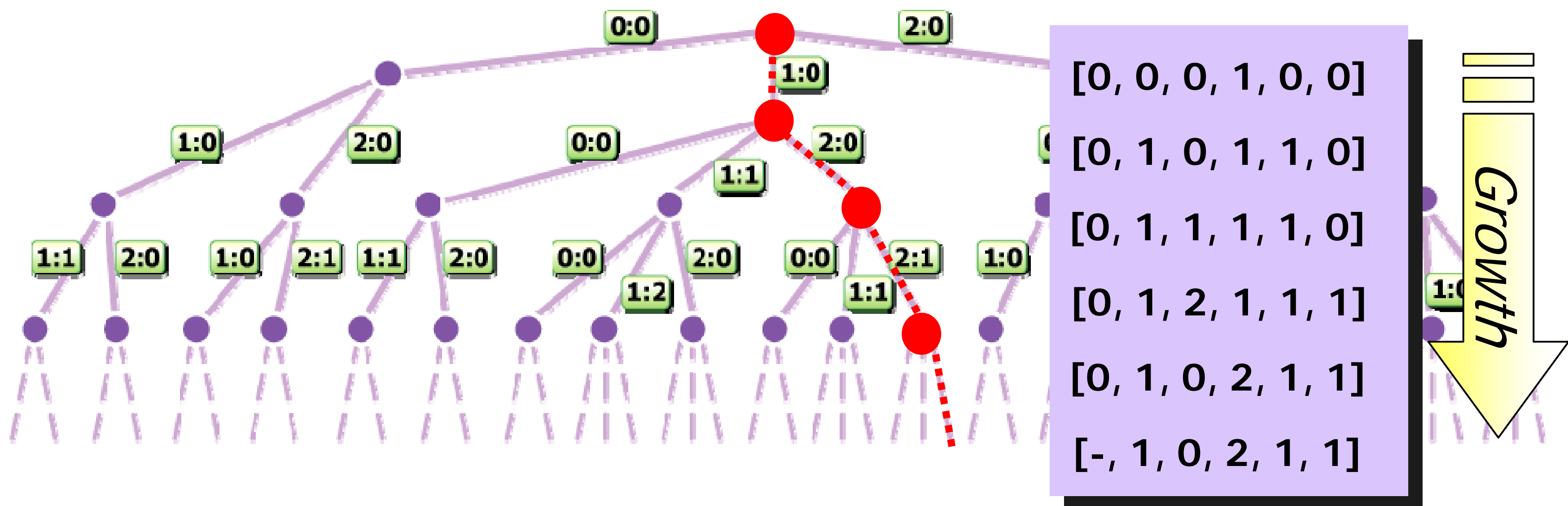

Creates information to tell us how to undo a transition

$seen := \{s_0\}$
$stack := [s_0]$
$DFS\,(s_0)$

$DFS\,(s)$
**for each** $\alpha \in enabled\,(s)$ **do**
$s' := \alpha\,(s)$
**if** $s' \notin seen$ **then**
$seen := seen \cup \{s'\}$
$push\,(stack, s')$
$DFS\,(s')$
$pop\,(stack)$
**end** $DFS$

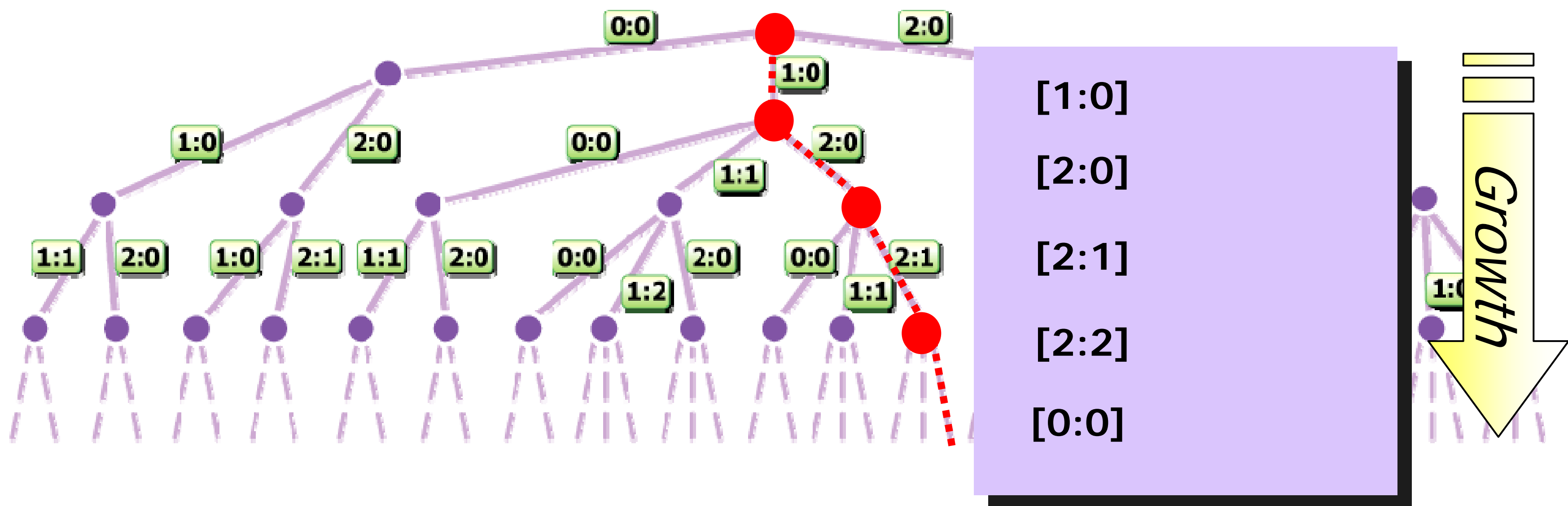
…customizable checking engine modules

# Depth-first Stack




- The depth-first stack can be implemented to hold state vectors
  - straight-forward implementation

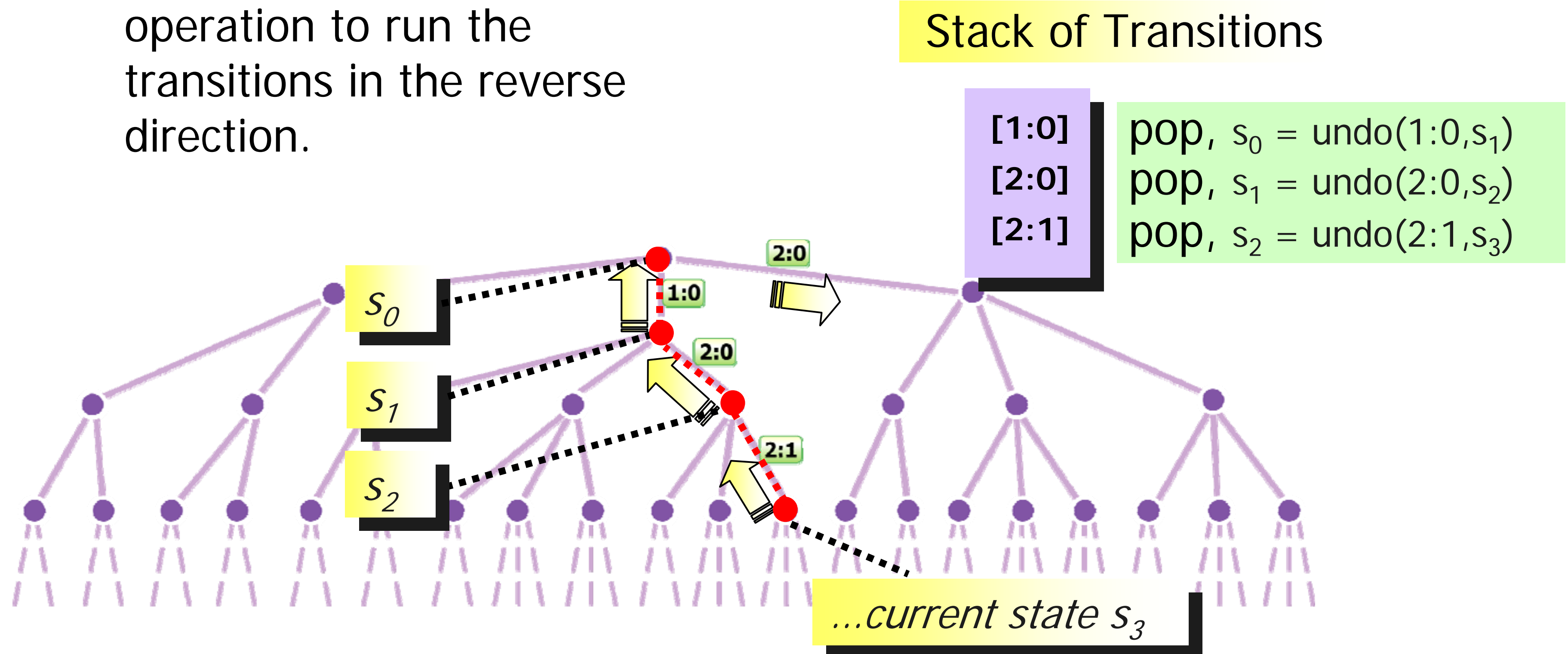
# Depth-first Stack




- The depth-first stack can be implemented to hold transitions
  - saves lots of space when working with real systems, but need the ability to "undo"

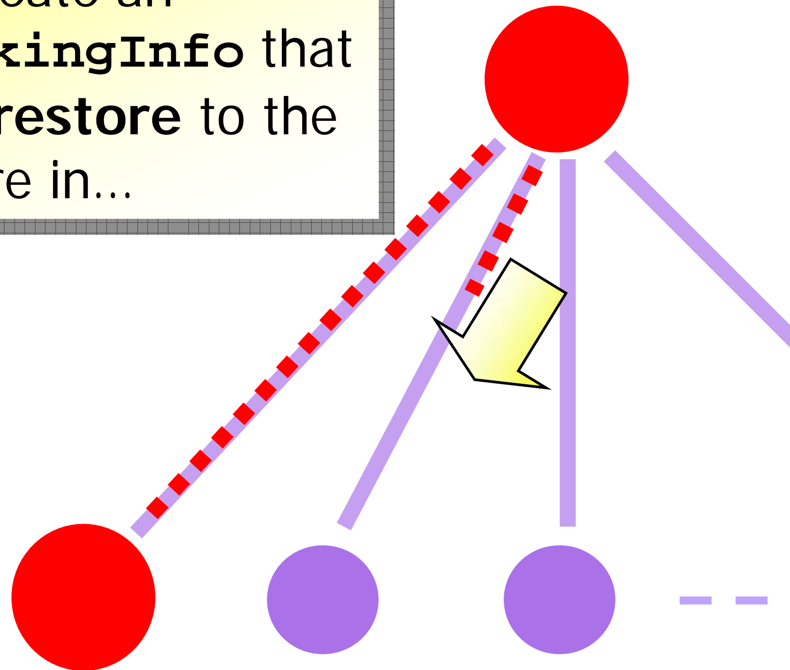
# Depth-first Stack of Transitions

- 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.



# Backtracking Information

When we are executing a transition, create an **`IBacktrackingInfo`** that tells how to **restore** to the state we were in…



BacktrackingInfo

Id of current state

Id of current thread

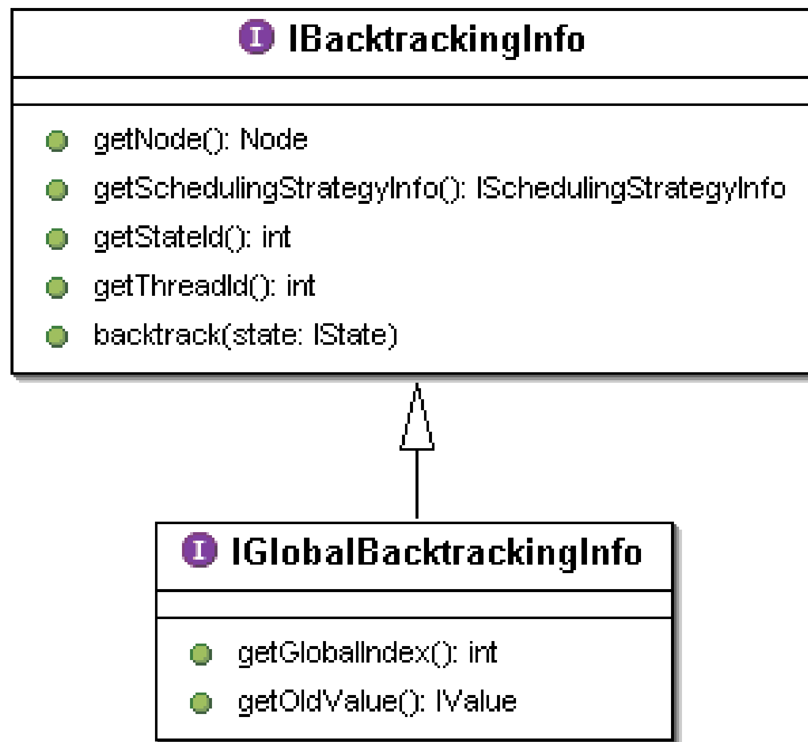
Transition Specific Information

e.g., Info about current value of variable being updated

ISchedulingStrategyInfo

- Info about total # of transitions
- Info about what to do next

# Backtracking Information




- Information needed to backtrack
  - state, thread ID, *etc.*
  - scheduling information
    - which non-deterministic choice was made, if any
  - specific info for each kind of action, transformation, etc.

# Backtracking Information – Code Example

- Backtracking an assignment to an int global variable:

```
class DefaultIntGlobalBacktrackingInfo … {
  private final int globalIndex;
  private final int v;
  private final Type vType;
  …
  public void backtrack(IState state) {
    state.setGlobalValue(
        globalIndex,
        vf.newIntValue(vType, v));
  }
}
```

# Backtracking Information – Code Example

- The action evaluator creates the backtracking info through the backtracking info factory:

```
public IGlobalBacktrackingInfo takeAssignGlobalAction(
        AssignAction a, int globalIndex, IValue value) {

  …
  IValue oldValue = s.getGlobalValue(globalIndex);
  s.setGlobalValue(globalIndex, value);

  return bif.createGlobalBacktrackingInfo(
            …, globalIndex, oldValue, ssi);
}
```

# SimpleSearcher.step()

A DFS implementation of the state-space exploration

*step* ()
  **if** *shouldBacktrack* () $\vee$ *isSeen* () $\vee$
      *hasNoActiveThreads* () **then**
    **return false**

  **if** *isInvalidEndState* () **then**
    *error* ( INVALID_END_STATE )
    **return false**

  $T$ := ss.*getEnabledTransformations* (s)

  *ssi* := ss.*newStrategyInfo* ()
  $\alpha$ := ss.*advise* (s, $T$, *ssi* )

  *push* (*newBacktrackingInfo* (s, $T$, $\alpha$, *ssi*))

  *doTransition* (s, $\alpha$, *ssi* )
  **return true**
**end** *step*

```
public class SimpleSearcher
       extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  IStateManager sm;
  void initialize() {
    s=createInitialState();
    sm.storeState(s);
  }
  void search() {
    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
}
```

# SimpleSearcher.step()

## A DFS implementation of the state-space exploration

```
step ()
    if shouldBacktrack () ∨ isSeen () ∨
            hasNoActiveThreads () then
        return false

    if isInvalidEndState () then
        error ( INVALID_END_STATE )
        return false

    T := ss.getEnabledTransformations (s)

    ssi := ss.newStrategyInfo ()
    α := ss.advise (s, T, ssi )

    push (newBacktrackingInfo (s, T, α, ssi))

    doTransition (s, α, ssi )
    return true
end step
```

…if we are forced to backtrack, we have visited the current state, or if all threads have completed, then we cannot step

…check if the current state is an invalid state (deadlock check)

…get all the enabled transformations by calling the ISchedulingStrategist

…call the ISchedulingStrategist to pick the next transition; *ssi* is used to record necessary information to make sure each transition will be executed eventually

…store the backtracking info necessary for reversing the transition, and for counter example generation

…execute the transition and return true to indicate a successful step

# SimpleSearcher.backtrack()

A DFS implementation of the state-space exploration

```
backtrack ()
  bi := pop ()

  while ¬bi.getSSI ().hasInfo () do
    bi.backtrack (s)
    if isStackEmpty ()
      return false
    bi := pop ()

  T := bi.getTransformations ()

  ssi := bi.getSSI ()
  α := ss.advise (s, T, ssi )

  push (newBacktrackingInfo (s, T, α, ssi))

  doTransition (s, α, ssi )
  return true
end backtrack
```

```
public class SimpleSearcher
       extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  IStateManager sm;
  void initialize() {
    s=createInitialState();
    sm.storeState(s);
  }
  void search() {
    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
}
```

# SimpleSearcher.backtrack()

## A DFS implementation of the state-space exploration

```
backtrack ()
    bi := pop ()

    while ¬bi.getSSI ().hasInfo () do
        bi.backtrack (s)
        if isStackEmpty ()
            return false
        bi := pop ()

    T := bi.getTransformations ()

    ssi := bi.getSSI ()
    α := ss.advise (s, T, ssi )

    push (newBacktrackingInfo (s, T, α, ssi))

    doTransition (s, α, ssi )
    return true
end backtrack
```

…get the last backtracking info

…keep backtracking until we find a state that is not fully expanded (i.e., all of its enabled transitions have not been explored); if it does not exist, then return false (i.e., all states have been fully expanded)

…get the enabled transformations

…call the ISchedulingStrategist to pick the next transition

…store the backtracking info necessary for reversing the transition, and for counter example generation

…execute the transition and return true to indicate a successful backtrack

# Outline

## Overview

- DFS algorithm
- Bogor components
- Configuration
- Initialization

## State Representation

- Types & Values
- Value Factory
- State Factory

## DFS Stack

- Search algorithm
- Scheduler

## Seen Before Set

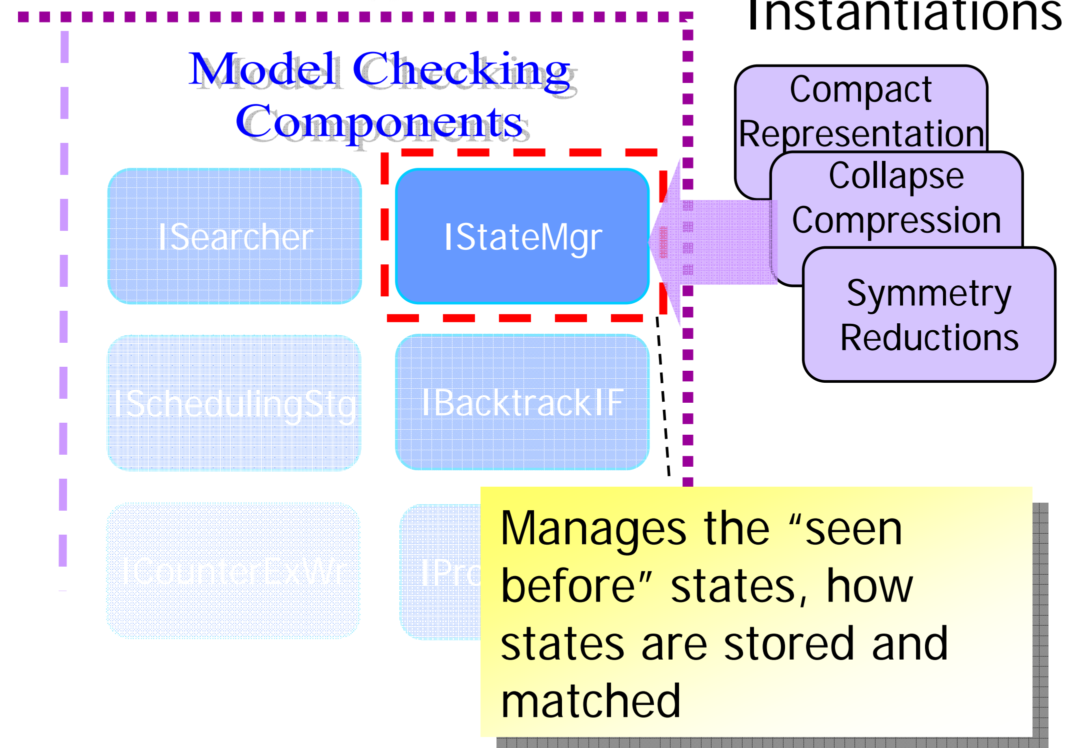
- State Manager

# Storing The State-space

## The `IStateManager` Module

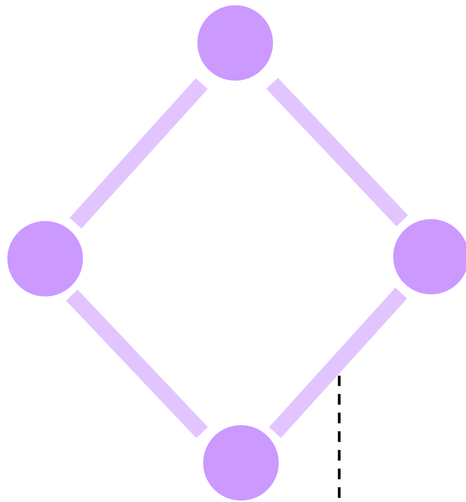
```
seen := {s0}
stack := [s0]
DFS (s0)

DFS (s)
  for each α ∈ enabled (s ) do
      s′ := α (s )
      if s′ ∉ seen then
         seen := seen ∪ {s′ }
         push (stack, s′ )
         DFS (s′ )
         pop (stack )
end DFS
```




...customizable checking engine modules

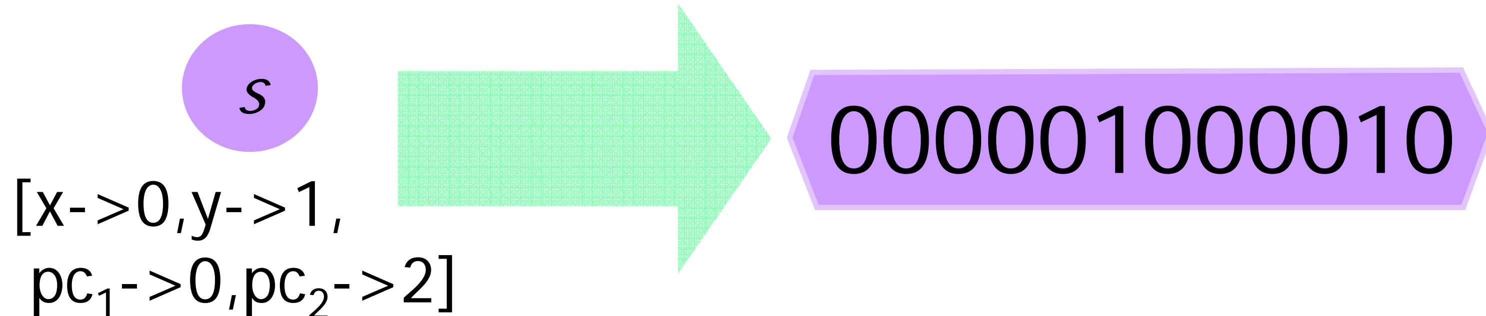
# Seen Before Set



We need to store states so we know when to backtrack

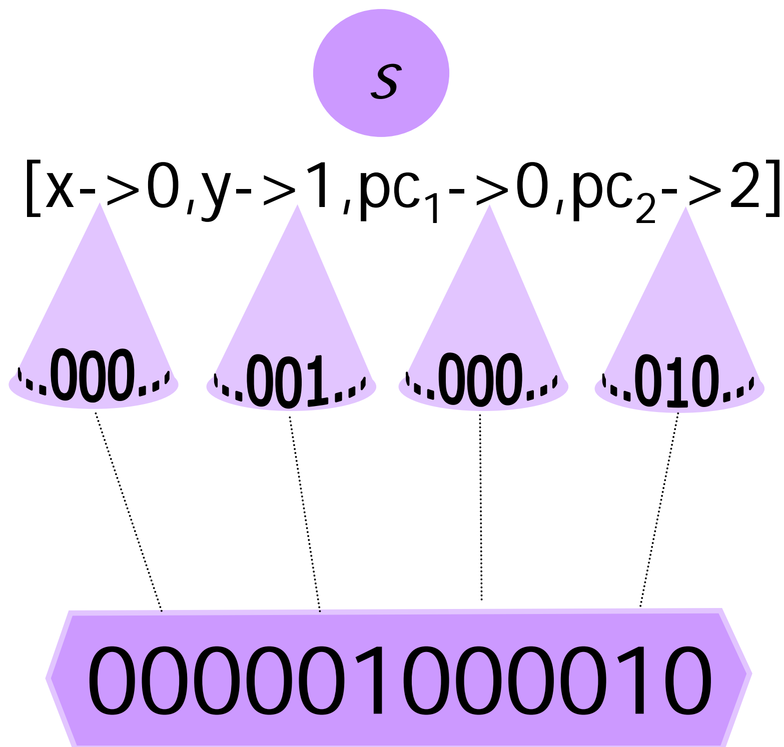
- Used to keep track states that have been visited before
  - *storing* the states
  - *matching* states
- In explicit state model checking, it is usually done by linearizing states to bit-vectors
  - store the bit-vectors in a set
  - match each new state bit vector against all stored ones
- Usually the *most computationally expensive* module
  - linearization and state matching takes some time
  - the number of states grow exponentially *wrt.* the system's complexity

# Seen Before Set




- To ensure exhaustive exploration, we need to preserve state equality, i.e.,
  - it should be able to generate the same bit-vector for states whose variables and PCs values are the same, and generate distinct bit-vectors for different ones
- Otherwise:
  - we may match distinct states
    - the model checker backtracks earlier than it should
    - errors may be missed (but errors found are real errors!)
  - we may not match equivalent states
    - redundant exploration of states and transitions

# Seen Before Set




- For space efficiency, we want to use the least number of bits possible to encode each value
- In basic BIR, we only have integer values, thus
  - each state can be represented by appending bit-vectors of variables and PCs values
  - we can compute the least number of bits possible from the type of each value

# Value Linearization

- For each integral type, we need $\lceil lg\ N \rceil$ bits, where *N* is the number of values the type can have, *e.g.*,
  - Each thread is represented by its *program counter*
    - *N* is number of locations in the model
  - Each variable is represented by its *value*
    - for each range int type, *N* is max – min + 1
      - value *x* is represented as *x* + min using *N* bits
    - for int type, *N* is $2^{32}$
    - for boolean type, N is 2

# Value Linearization – Code Example

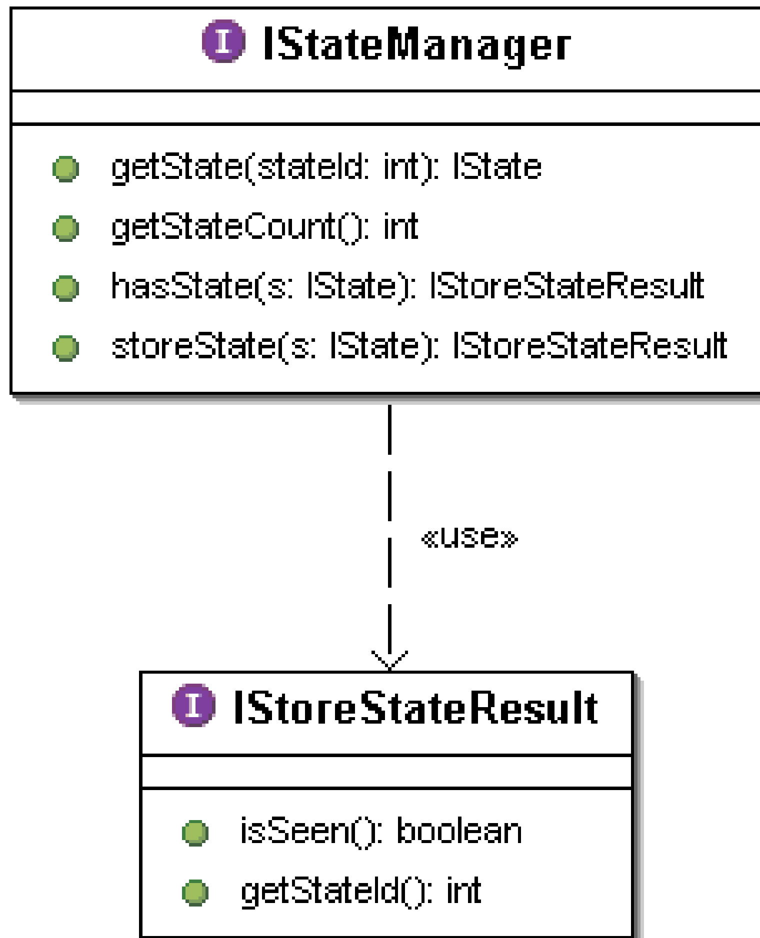
- To linearize an int range type:

```
int value = …;
IntRangeType irt = …;
BitBuffer bb = …;

int highLimit = irt.getHighLimit();
int lowLimit = irt.getLowLimit();
int count = highLimit - lowLimit + 1;
bitLength = Util.widthInBits(count - 1);
value -= lowLimit;

bb.append(value, bitLength);
…
```

# IStateManager




- Used to keep track states
- Also assign a unique number for each stored state (stateId)
  - use the number instead of the actual state in the DFS stack

# SimpleStateManager – Code Example

- To store a state, we need to linearize it, then put it in a table that maps it to a unique integer:

```
public IStoreStateResult storeState(IState s) {
  boolean seen = true;
  int id = 0;
  …
  StaticByteArray o = linearize(s);
  id = stateStateIdTable.get(o);

  if (id == 0) {
    id = nextStateId++;
    stateStateIdTable.put(o, id);
    seen = false;
  }
  return …
}
```

# Assessment

- Bogor architecture is highly modular
  - clean API using design patterns
  - customizable components allows easy incorporation of targeted algorithms for particular family of software artifacts