

# Software Model Checking Using Bogor

- a Modular and Extensible Model Checking Framework

*3rd Estonian Summer School in  
Computer and System Science (ESSCaSS'04)*

Slide Set 03: Bogor Architecture

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

<http://www.cis.ksu.edu/~hatcliff/ESSCaSS04>

---

**John Hatcliff**

Matthew B. Dwyer

Robby

*SAnToS Laboratory, Kansas State University, USA*

## 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

# Core DFS Algorithm

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

...put initial state in seen set

...current path being explored in computation tree begins with initial state

```
DFS ()
```

...iterate if there exists an unexpanded state

```
while stack ≠ ∅ do
```

...current state to expand

```
  s := pop(stack)
```

```
  workSet := enabled(s)
```

...get the transitions to explore at this state

```
  for each  $\alpha \in \textit{workSet}$  do
```

...for each transition

```
    s' :=  $\alpha$ (s)
```

...calculate the successor state

```
    if  $s' \notin \textit{seen}$  then
```

```
      seen := seen ∪ {s'}
```

...if *s'* has not been seen before, then put it in the seen set

```
      push(stack, s')
```

```
end DFS
```

...put *s'* on the stack which represents the current path in the tree

# Motivations and Goals

- If you're going to customize Bogor, you need to understand what is going on in the guts of the code.
- We've learned the basic DFS algorithm, and the basic data structures, now we are going to learn how they are implemented in Bogor.
- We'll survey the architecture of Bogor
  - many details will be omitted
  - for more info, see the API documentation and the code!

# Outline



## ● Overview

- Bogor components
- Configuration
- Initialization

## State Representation

- Types & Values
- Value Factory
- State Factory

## DFS Stack

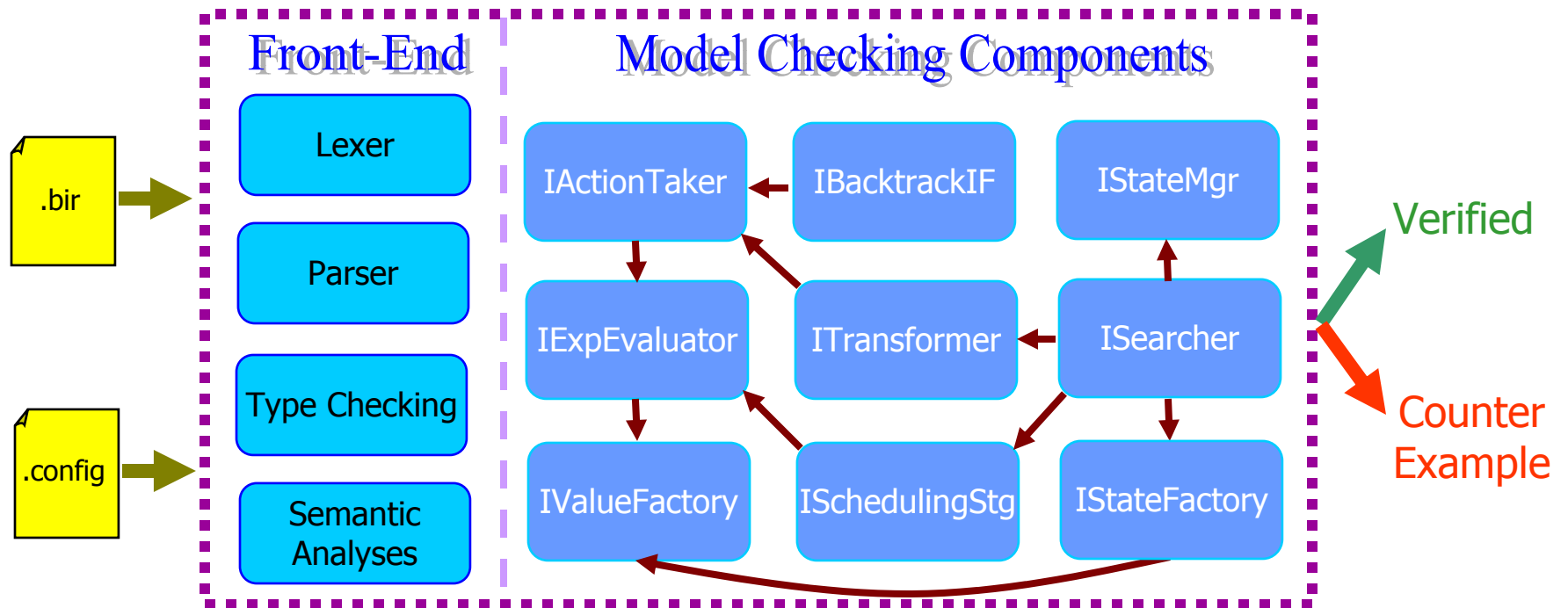
- Search algorithm
- Scheduler

## Seen Before Set

- State Manager

# Bogor Architecture

## Bogor -- Customizable Checking Engine Modules



*...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

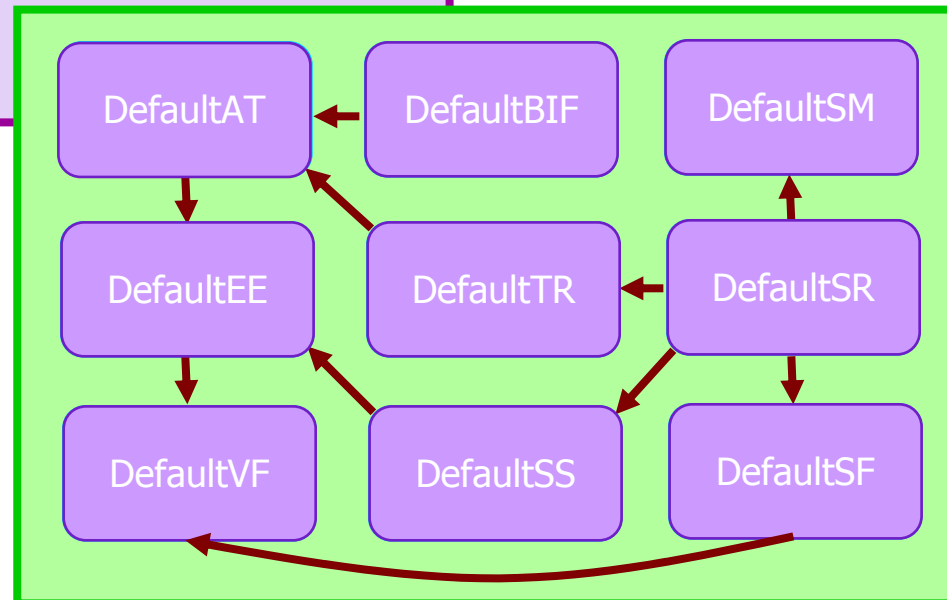
# Bogor Initialization

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

ISearcher.maxErrors   = 1
...
```

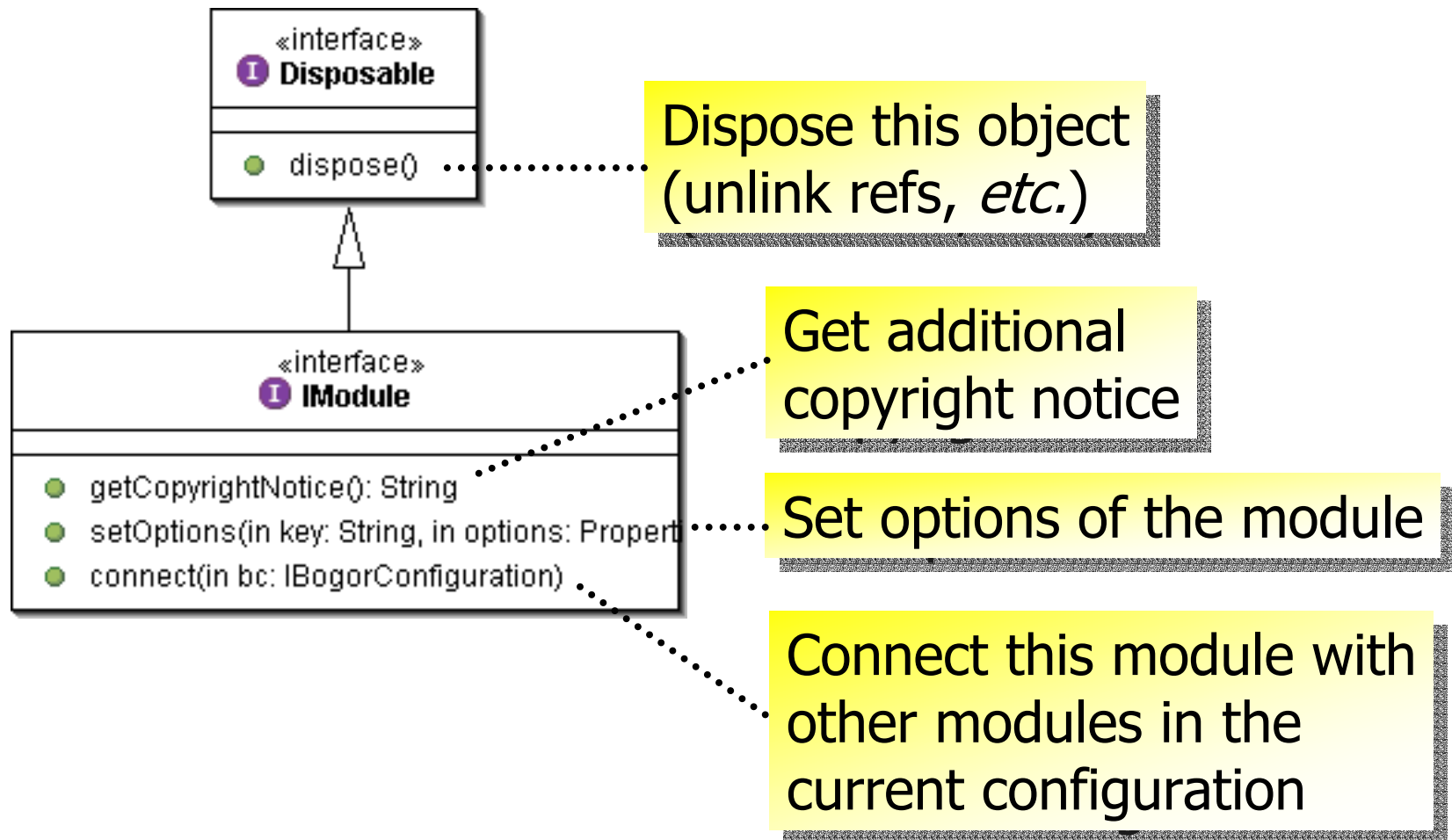
Given a configuration, Bogor instantiate the specified components

Options are passed to each component, and connections are established



# Bogor Module Interface

Each Bogor component must implement IModule



# Outline



## ● Overview

- Bogor components
- Configuration
- Initialization

## State Representation

- Types & Values
- Value Factory
- State Factory

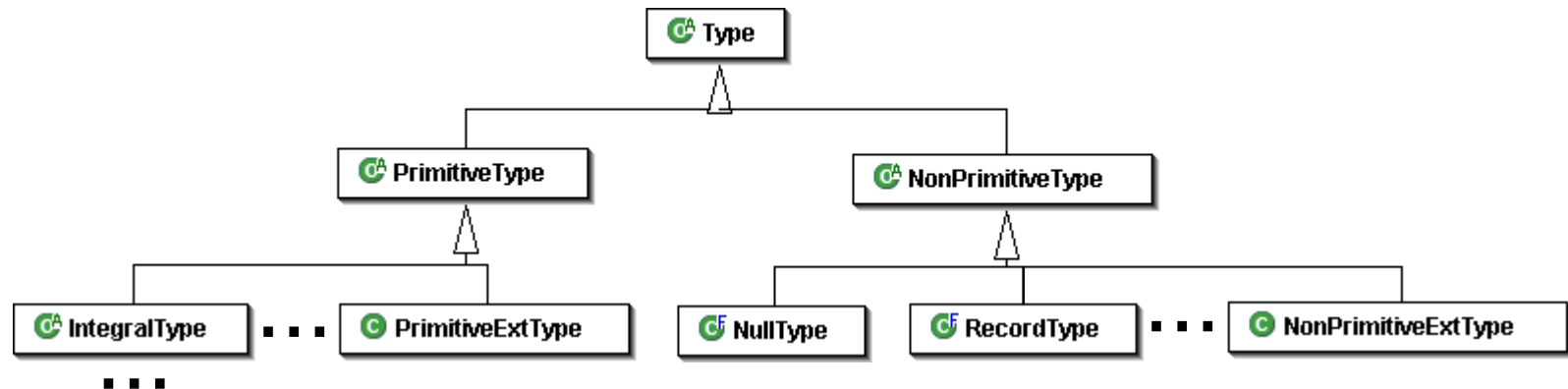
## DFS Stack

- Search algorithm
- Scheduler

## Seen Before Set

- State Manager

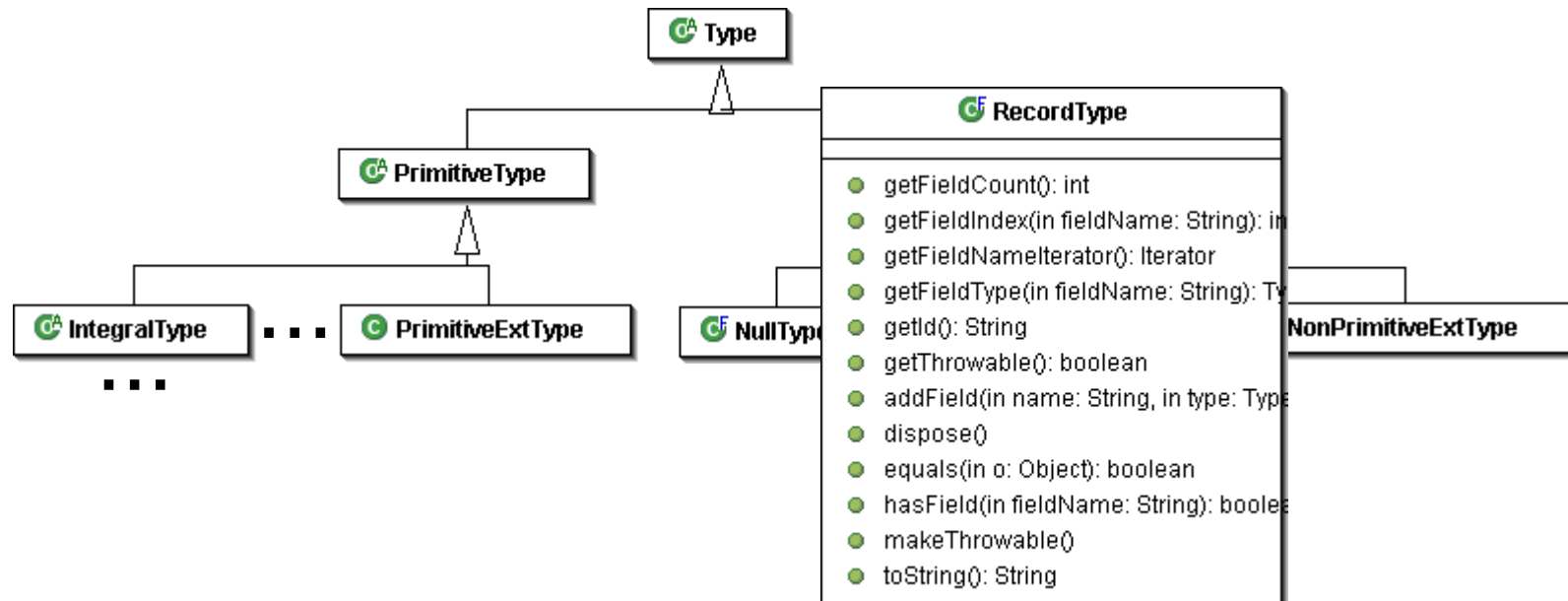
# Type and Value Representations



- 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 represented

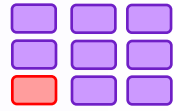
*Package:* bogor.type

# Type and Value Representations

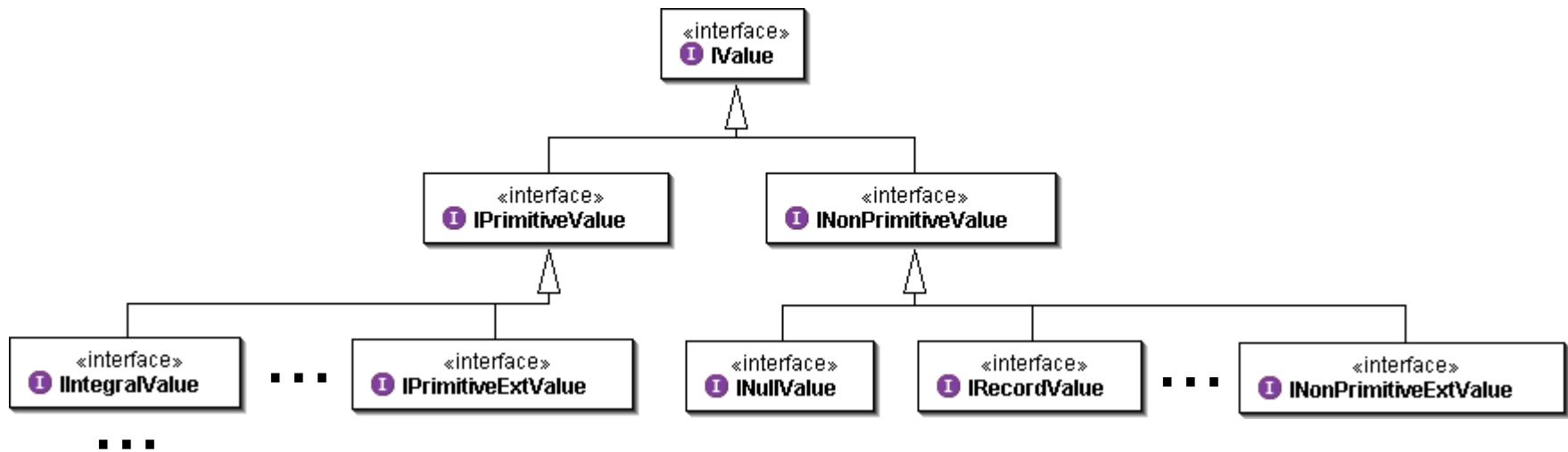


For example, the record type class has methods to access its fields's names, types, and indices

*Package:* bogor . type

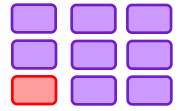


# Type and Value Representations

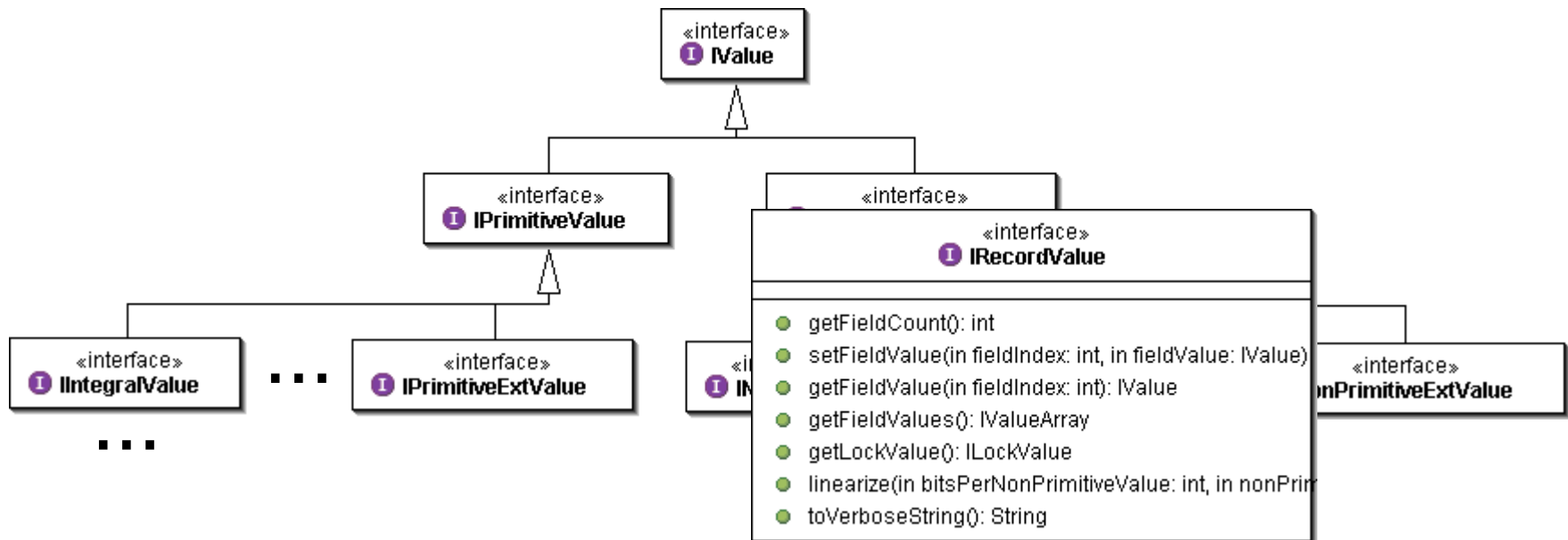


- BIR values mimic BIR types structure
  - Primitive values: int, long, float, double, *etc.*
  - Non-primitive values: null, record, lock, *etc.*
- Values are created using a ValueFactory

*Package:* bogor.module.value

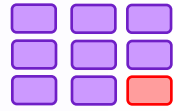


# Type and Value Representations

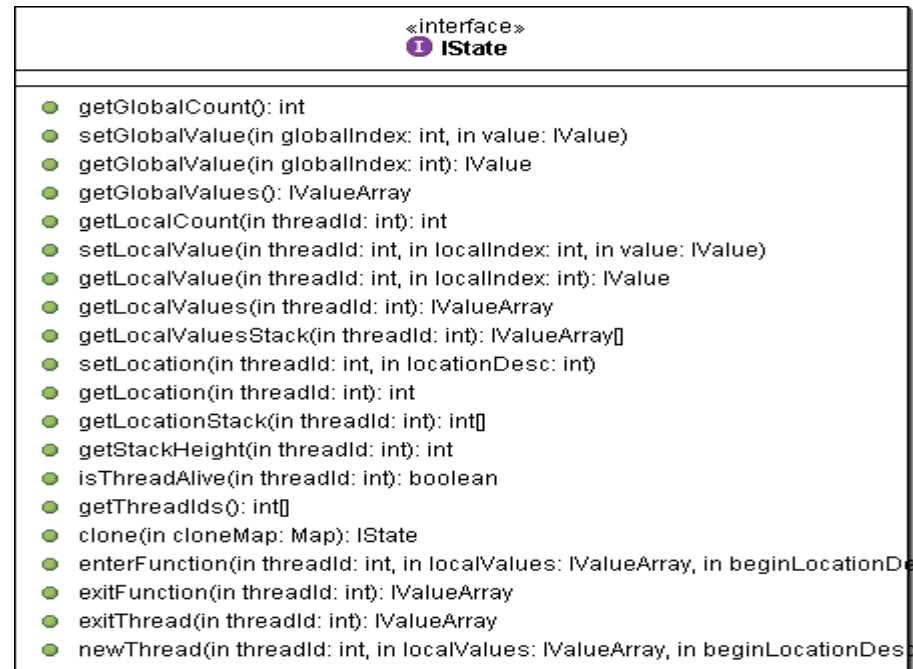


Similar to the record type, the record value interface has methods to access its fields' values

*Package:* `bogor.module.value`



# State Representation



- 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

- Bogor components
- Configuration
- Initialization

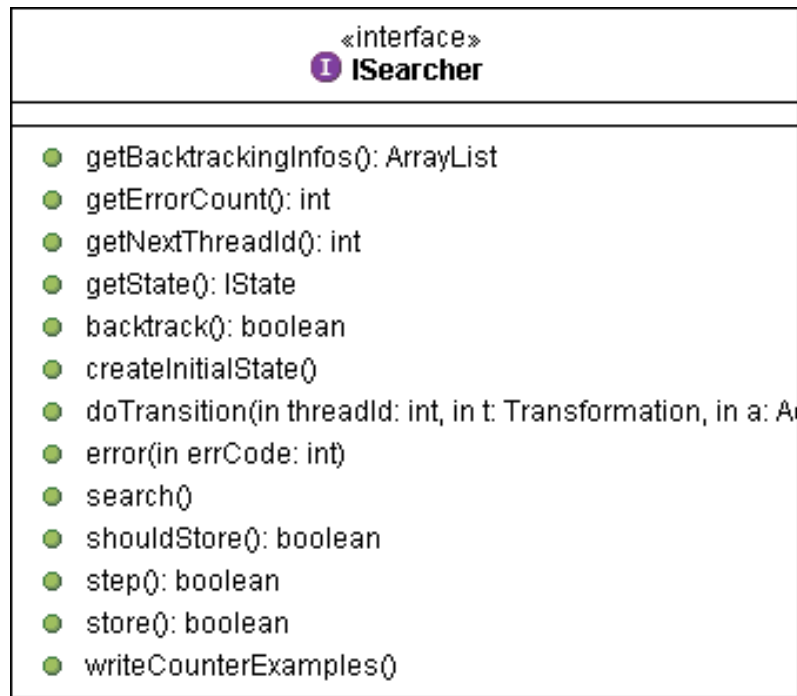
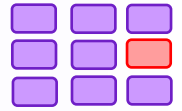
## DFS Stack

- Search algorithm
- Scheduler

## Seen Before Set

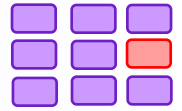
- State Manager

# Search Module



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

*Package:* bogor.module

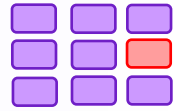


# DefaultSearcher

A DFS implementation of the state-space exploration

```
seen := {s0}  
stack := [s0]  
DFS ()  
  
DFS ()  
  while stack ≠ ∅ do  
    s := pop (stack)  
    workSet := enabled (s)  
    for each  $\alpha \in$  workSet do  
      s' :=  $\alpha$  (s)  
      if s' ∉ seen then  
        seen := seen ∪ {s'}  
        push (stack, s')  
  
end DFS
```

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



# DefaultSearcher.step()

A DFS implementation of the state-space exploration

```
step ()
  if shouldBacktrack () ∨ isSeen () ∨
    hasNoRunnableThreads () 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)

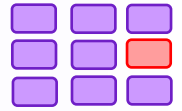
  push (newBacktrackingInfo (s, T, α, ssi))

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

```
public class DefaultSearcher
  extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  void search() {

    s=createInitialState();

    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
  ...
}
```



# DefaultSearcher.step()

## A DFS implementation of the state-space exploration

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

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

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

...check if the current state is an invalid state

```
  T := ss.getEnabledTransformations (s)
```

...get all the enabled transformations by calling the ISchedulingStrategist

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

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

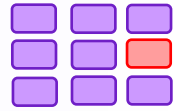
```
  push (newBacktrackingInfo (s, T,  $\alpha$ , ssi))
```

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

```
  doTransition (s,  $\alpha$ , ssi)  
  return true
```

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

```
end step
```



# DefaultSearcher.backtrack()

A DFS implementation of the state-space exploration

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

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

  T := bi.getTransformations()

  ssi := bi.getSSI()
   $\alpha$  := ss.advise(s, T, ssi)

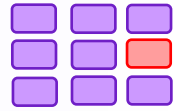
  push(newBacktrackingInfo(s, T,  $\alpha$ , ssi))

  doTransition(s,  $\alpha$ , ssi)
  return true
end backtrack
```

```
public class DefaultSearcher
  extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  void search() {

    s=createInitialState();

    while (true) {
      if (!step()) {
        if (!backtrack()) {
          break;
        }
      }
    }
  }
  ...
}
```



# DefaultSearcher.backtrack()

A DFS implementation of the state-space exploration

```
backtrack ()
```

```
  bi := pop ().....
```

...get the last backtracking info

```
  while  $\neg$ bi.getSSI ().hasInfo () do :
```

```
    bi.backtrack (s)
```

```
    if isStackEmpty ()
```

```
      return false
```

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

...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)

```
  T := bi.getTransformations ().....
```

...get the enabled transformations

```
  ssi := bi.getSSI ()
```

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

...call the ISchedulingStrategist to pick the next transition

```
  push (newBacktrackingInfo (s, T,  $\alpha$ , ssi)).....
```

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

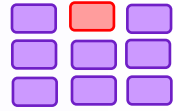
```
  doTransition (s,  $\alpha$ , ssi)
```

```
  return true
```

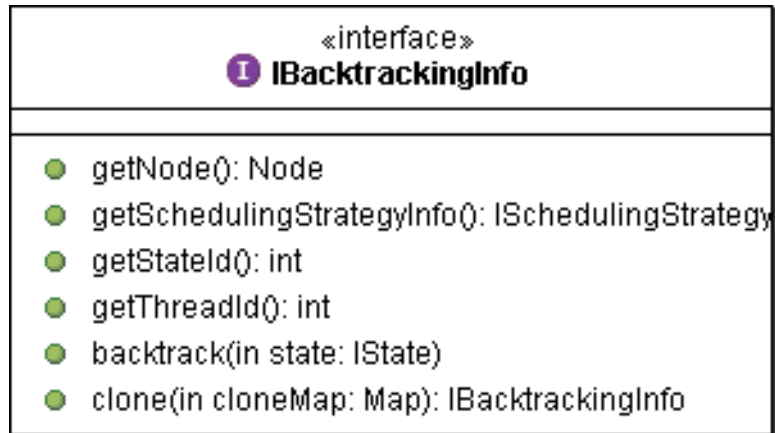
```
end backtrack
```

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

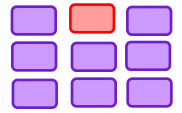




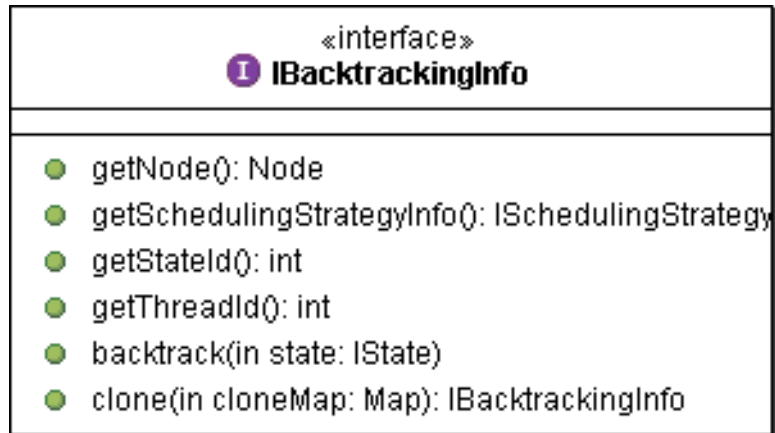
# Backtracking Information



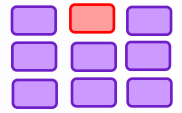
- Used to keep track information for
  - “undo”-ing transition
  - scheduling information for counter-example generation



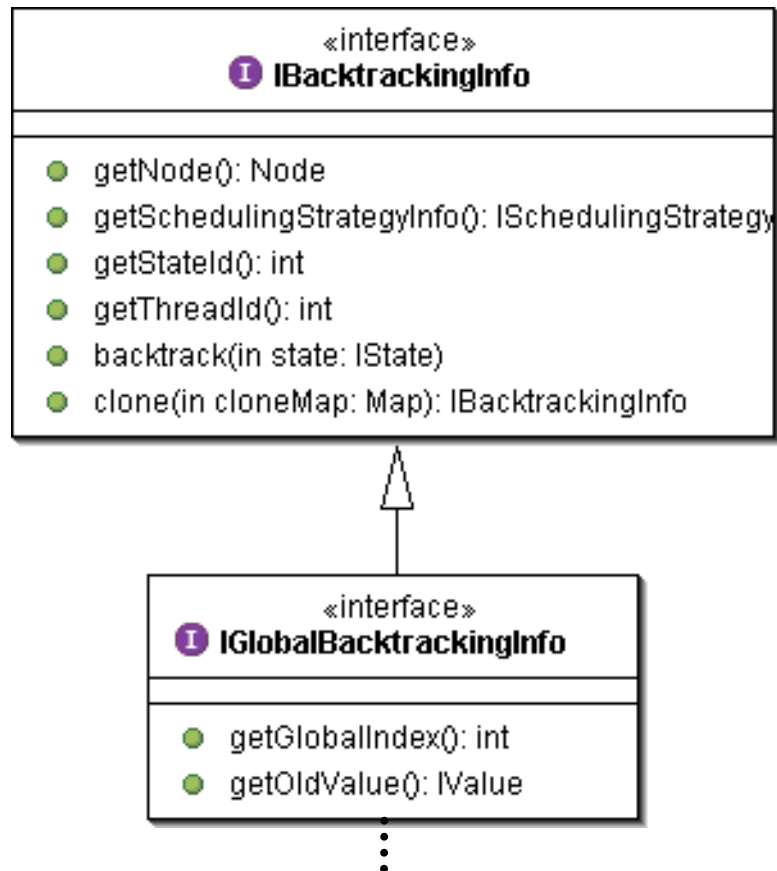
# Backtracking Information



- 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

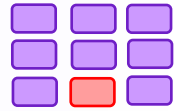


...for backtracking a global variable assignment, we need the index of global variable and its value before the assignment

- 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 Example

- Show simple assignment to global, and then annotations to show states and backtracking info.

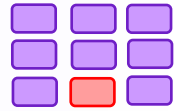


# Scheduler

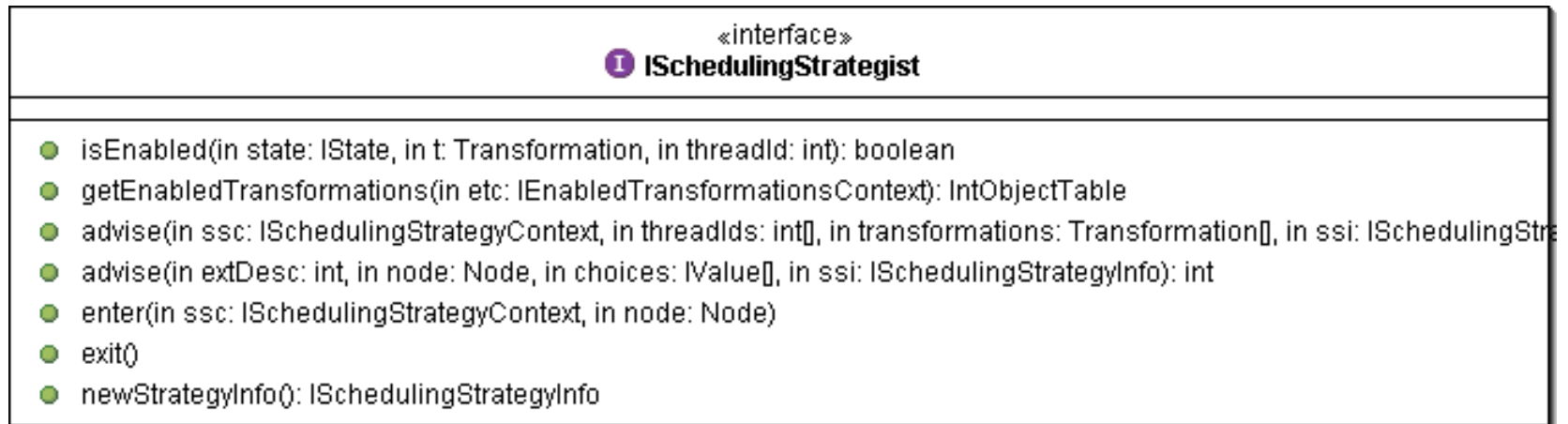
```
seen := {s0}  
stack := [s0]  
DFS ()
```

```
DFS ()  
  while stack ≠ ∅ do  
    s := pop (stack)  
    workSet := enabled (s)  
    for each  $\alpha \in \textit{workSet}$  do  
      s' :=  $\alpha$  (s)  
      if s' ∉ seen then  
        seen := seen ∪ {s'}  
        push (stack, s')  
end DFS
```

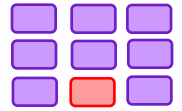
- Used to determine
  - enabled transitions
  - which transition to take



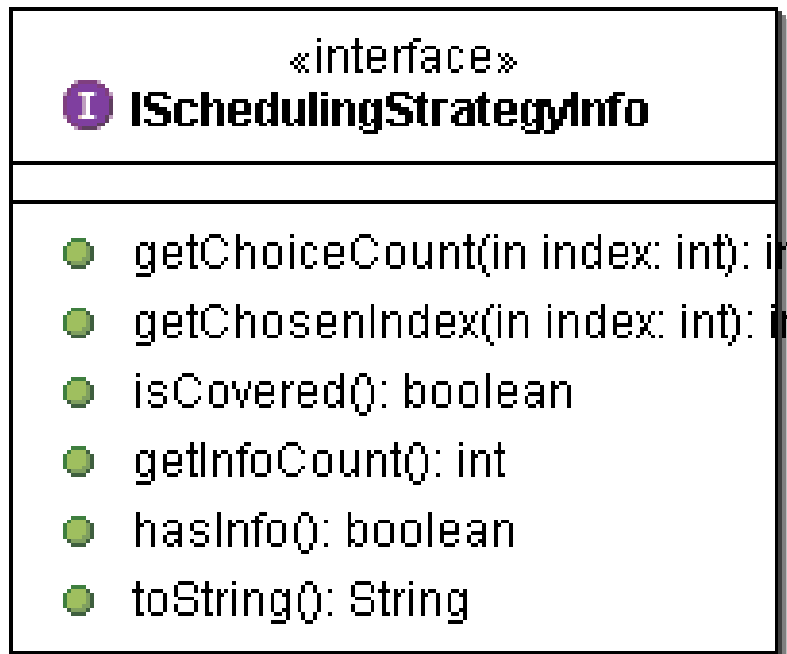
# ISchedulingStrategist



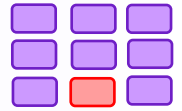
- Used to determine
  - enabled transitions: isEnabled(), getEnabledTransformations()
  - which transition to take: advise()
- create strategy info



# ISchedulingStrategyInfo



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

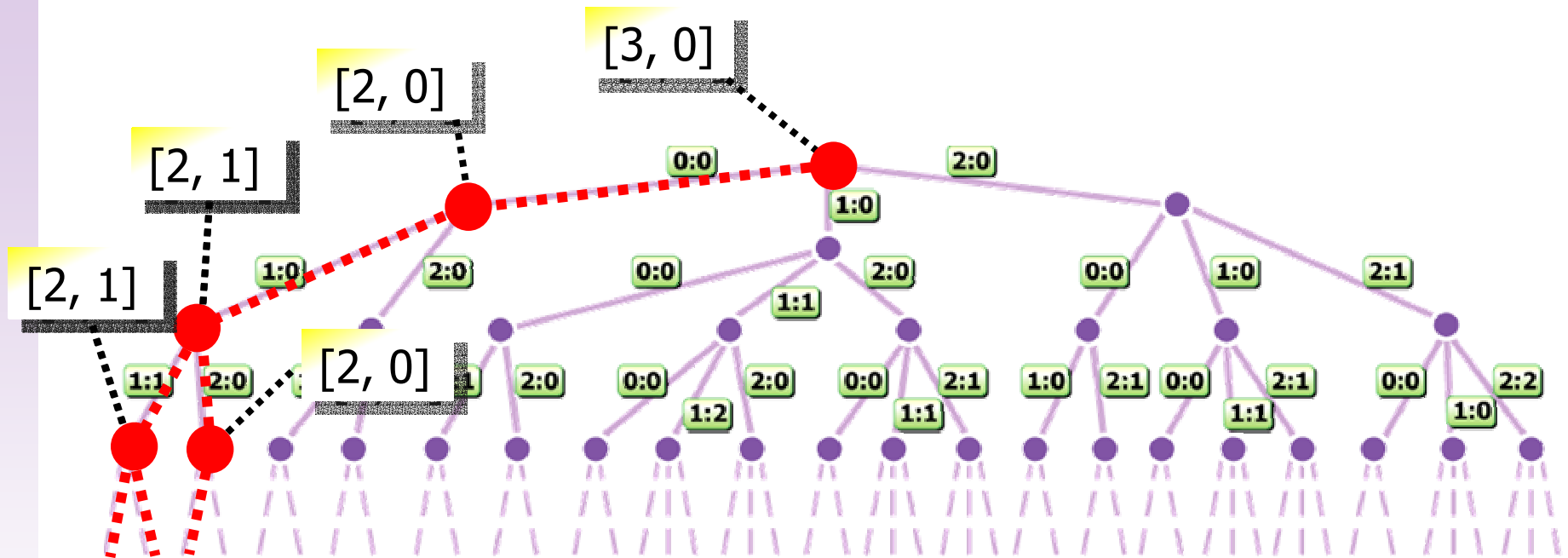
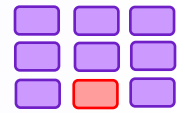


# DefaultSchedulingStrategist

C DefaultSchedulingStrategist	
● <sup>C</sup>	DefaultSchedulingStrategist()
●	getCopyrightNotice(): String
●	isEnabled(in state: IState, in t: Transformation, in threadId: int): boolean
●	getEnabledTransformations(in etc: IEnabledTransformationsContext): IntObjectTable
●	setOptions(in key: String, in configuration: Properties)
●	advise(in ssc: ISchedulingStrategyContext, in threadIds: int[], in transformations: Transformation[], in ssi: ISchedulingStrategyInfo)
●	connect(in bc: IBogorConfiguration)
●	dispose()
●	enter(in ssc: ISchedulingStrategyContext, in node: Node)
●	exit()
●	newStrategyInfo(): ISchedulingStrategyInfo

- 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

# DefaultSchedulingStrategist & DefaultSchedulingStrategyInfo



[number of enabled transitions, last index chosen]

# Outline



## State Representation

- Types & Values
- Value Factory
- State Factory

## Overview

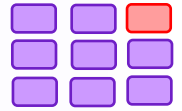
- Bogor components
- Configuration
- Initialization

## ● DFS Stack

- Search algorithm
- Scheduler

## Seen Before Set

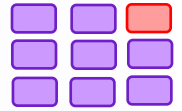
- State Manager



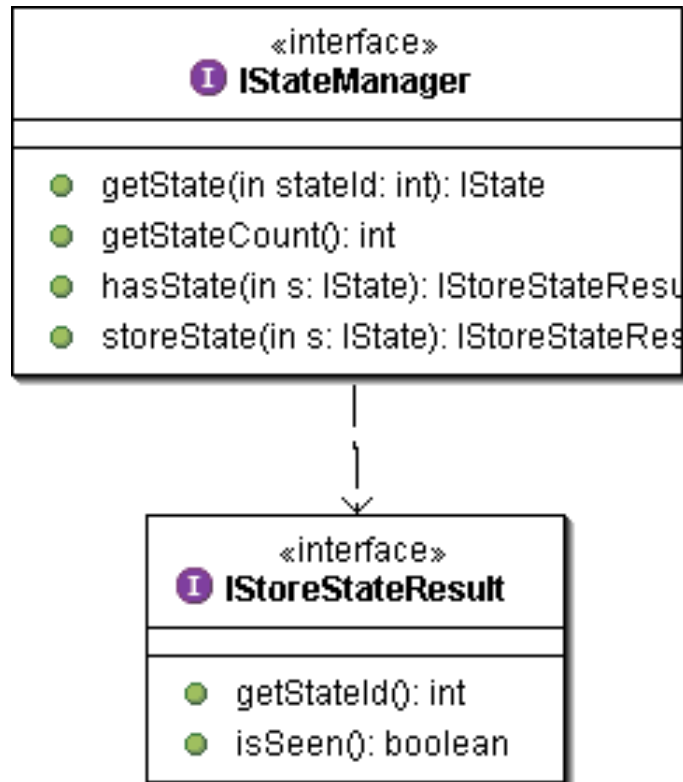
# Seen Before Set

```
seen := {s0}  
stack := [s0]  
DFS()  
  
DFS()  
  while stack ≠ ∅ do  
    s := pop(stack)  
    workSet := enabled(s)  
    for each  $\alpha \in \textit{workSet}$  do  
      s' :=  $\alpha$ (s)  
      if s' ∉ seen then  
        seen := seen ∪ {s'}  
        push(stack, s' )  
  end DFS
```

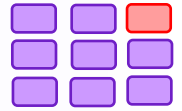
- Used to keep track states that have been visited before
- Usually done by linearizing the state into a bit-vector
- For efficiency,
  - store the state using the least number of bits possible (fingerprinting)
- For full state-space exploration
  - should preserve state equality (analogous to loss-less compressions)



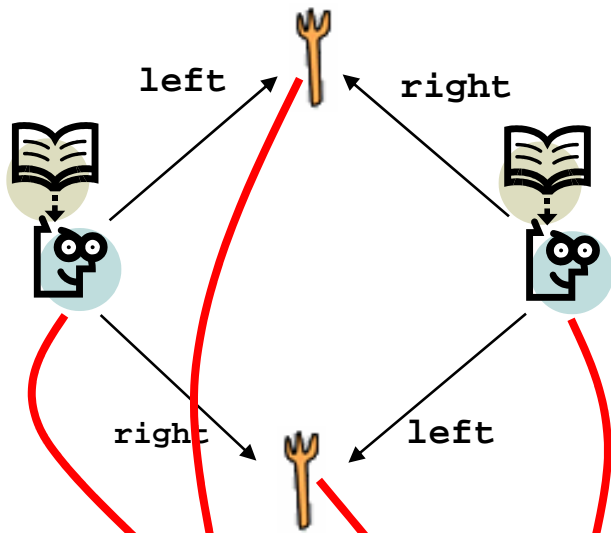
# IStateManager



- 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



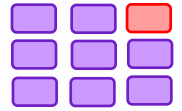
# State Linearization



```
[ Threads [ Philosopher1.loc0[Fork#1, Fork#2],  
            Philosopher2.loc0[Fork#2, Fork#1] ],  
  Heap[ Fork#1[false], Fork#2[false] ] ]
```

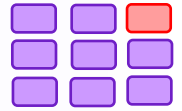
```
<Threads[ 000[0.1], 000[1.0] ],  
  Heap[ 1[0], 1[0] ]>
```

- Bogor states consist of
  - global variables
  - heap (consisting non-primitive values)
  - thread stores
- ...represented as shape graphs
- To efficiently compare them, we convert them to bit-vectors
  - each primitive value is converted to its bit-vector representation
    - want to make sure we use the least number of bits possible

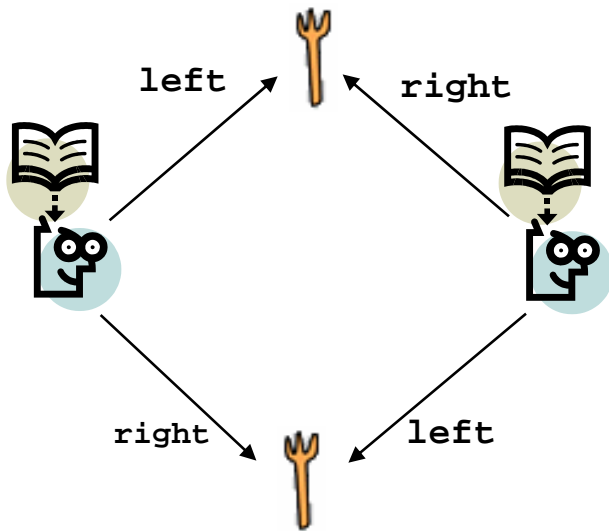


# State Linearization

- For each integral type, we need  $\text{ceil}(\lg N)$  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 ranged integer,  $N$  is  $\text{max} - \text{min} + 1$ 
      - value  $x$  is represented as  $x + \text{min}$  using  $N$  bits
    - for enumeration,  $N$  is the number of enumeration elements



# State Linearization



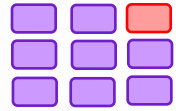
For the `TwoDiningPhilosophers.bir` example

- there are 5 locations, thus, we need 3 bits
  - the first declared location is numbered 0, and
  - the last declared location is numbered 4
- each fork is boolean, thus, we need 1 bit for each
- suppose we use 1 bit to encode object refs
- suppose we use 1 bit to encode record type
- The initial state

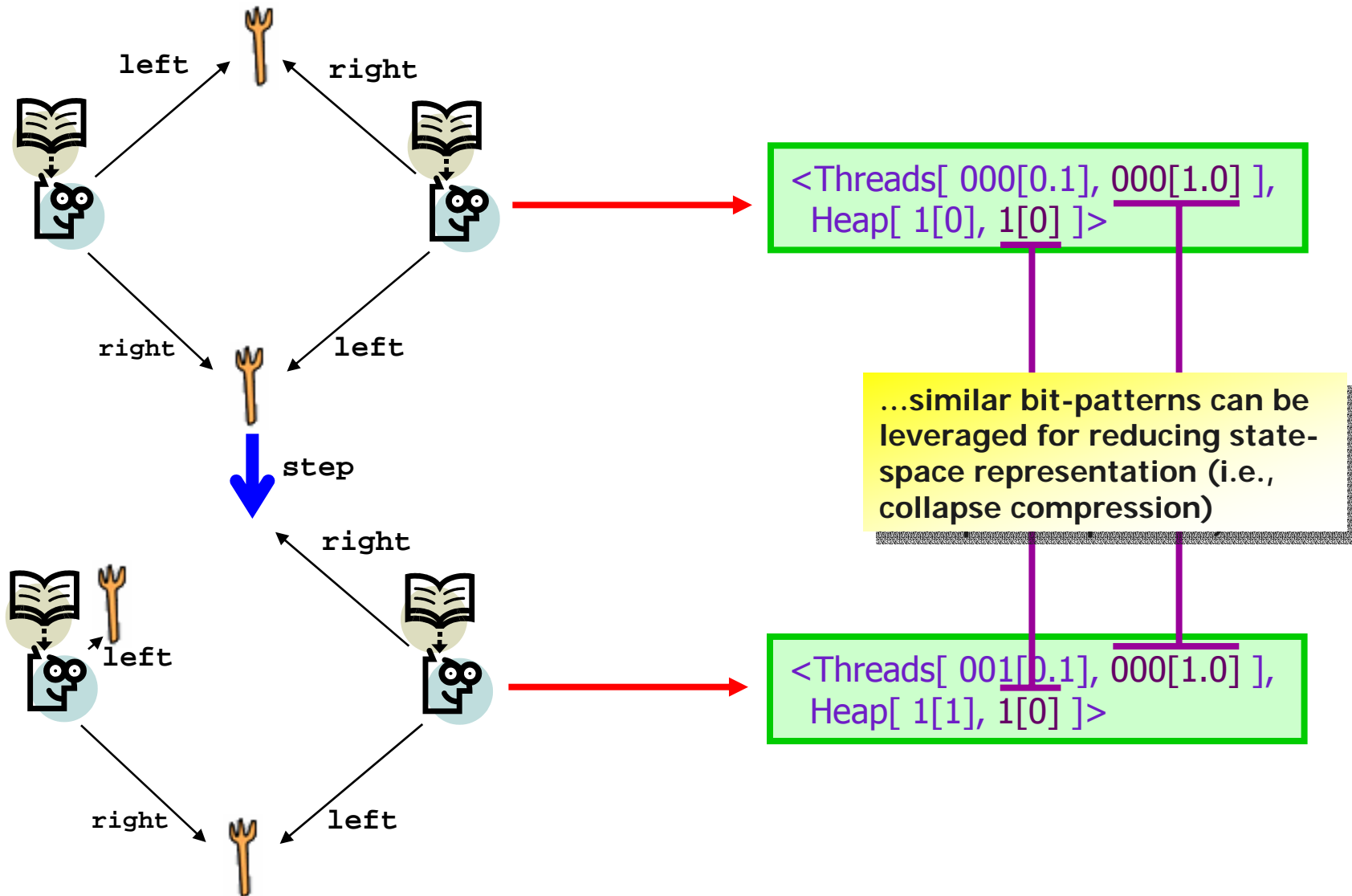
```
[ Threads [ Philosopher1.loc0[Fork#1, Fork#2],  
            Philosopher2.loc0[Fork#2, Fork#1] ],  
  Heap[ Fork#1[false], Fork#2[false] ] ]
```

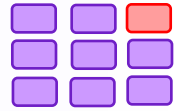
is represented as the bit vector

```
<Threads[ 000[0.1], 000[1.0] ],  
  Heap[ 1[0], 1[0] ]>
```

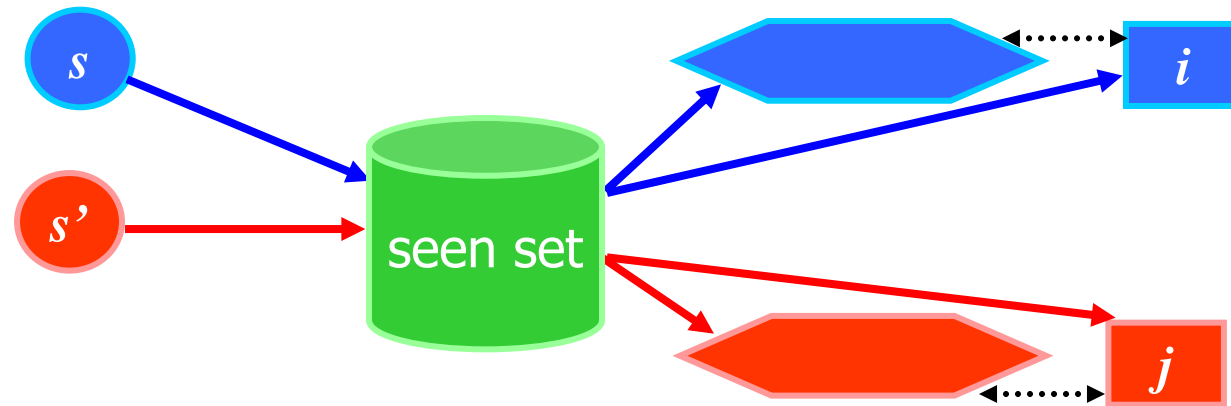


# State Linearization





# DefaultStateManager



- Convert each state to a bit-vector
- Use a table to map each state to its unique integer
- If a new state is stored, its unique number is the number of states stored so far

# 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