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 I: Bogor Overview

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

Goals of This STRESS Lecture

- Introduction to the primary features/functions of Bogor
- Overview of Bogor APIs that will allow you to easily modify Bogor or to add new functionality
  - requires some effort to learn
  - ...but a number of people have already implemented Bogor extensions
- Overview of some of the more sophisticated customizations of Bogor
- Get feedback from you as to what features/support you might want to have in Bogor that would make it more useful for you

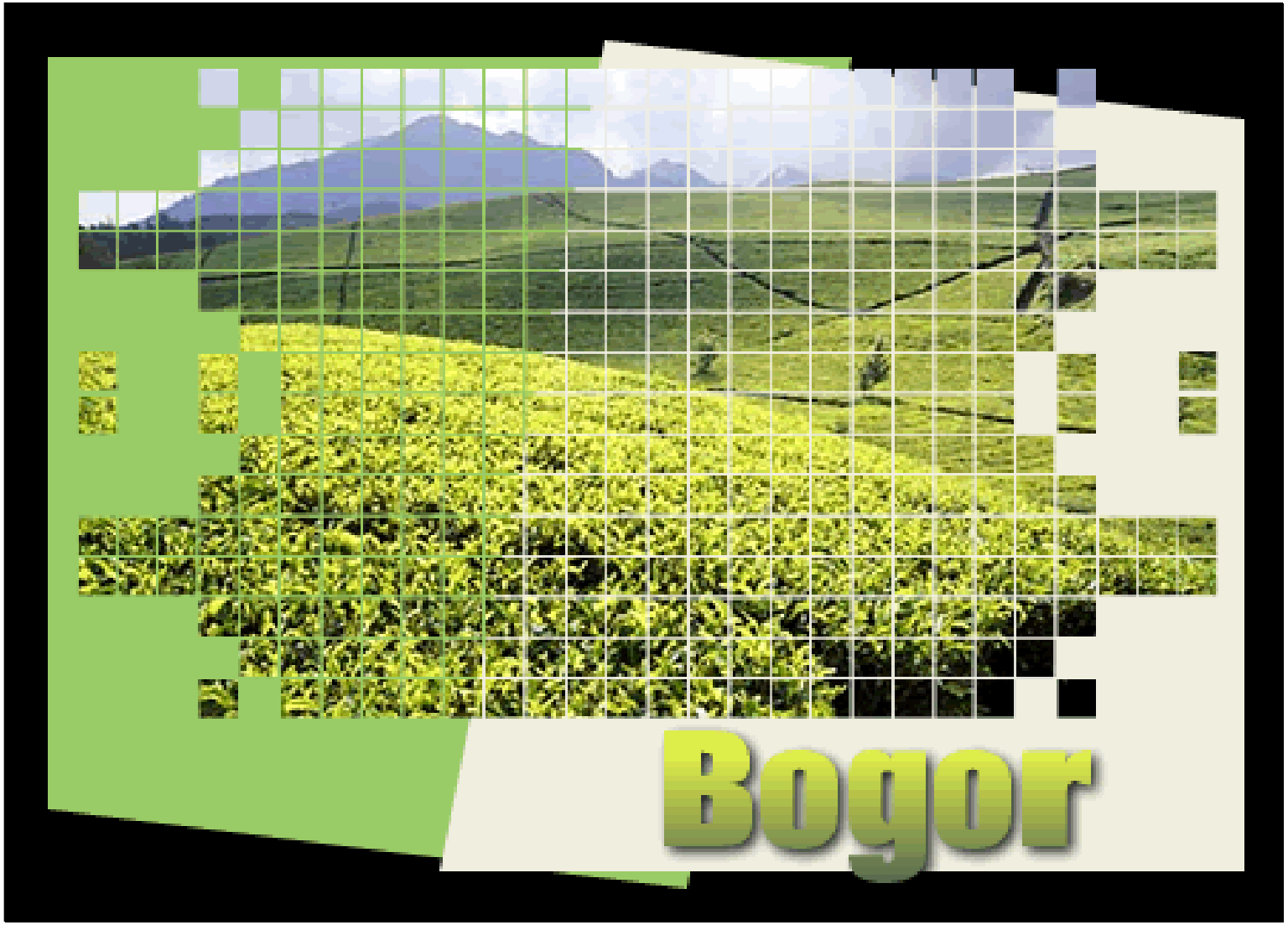
Lecture Organization

- Lecture I: Bogor Overview
  - Bogor Features
  - Motivation
  - BIR Overview
  - Bogor UI Demo
- Lecture II: DFS in Explicit State Model Checking
  - schedules and traces
  - computation tree
  - exhaustive search using DFS
  - DFS basic data structures
- Break

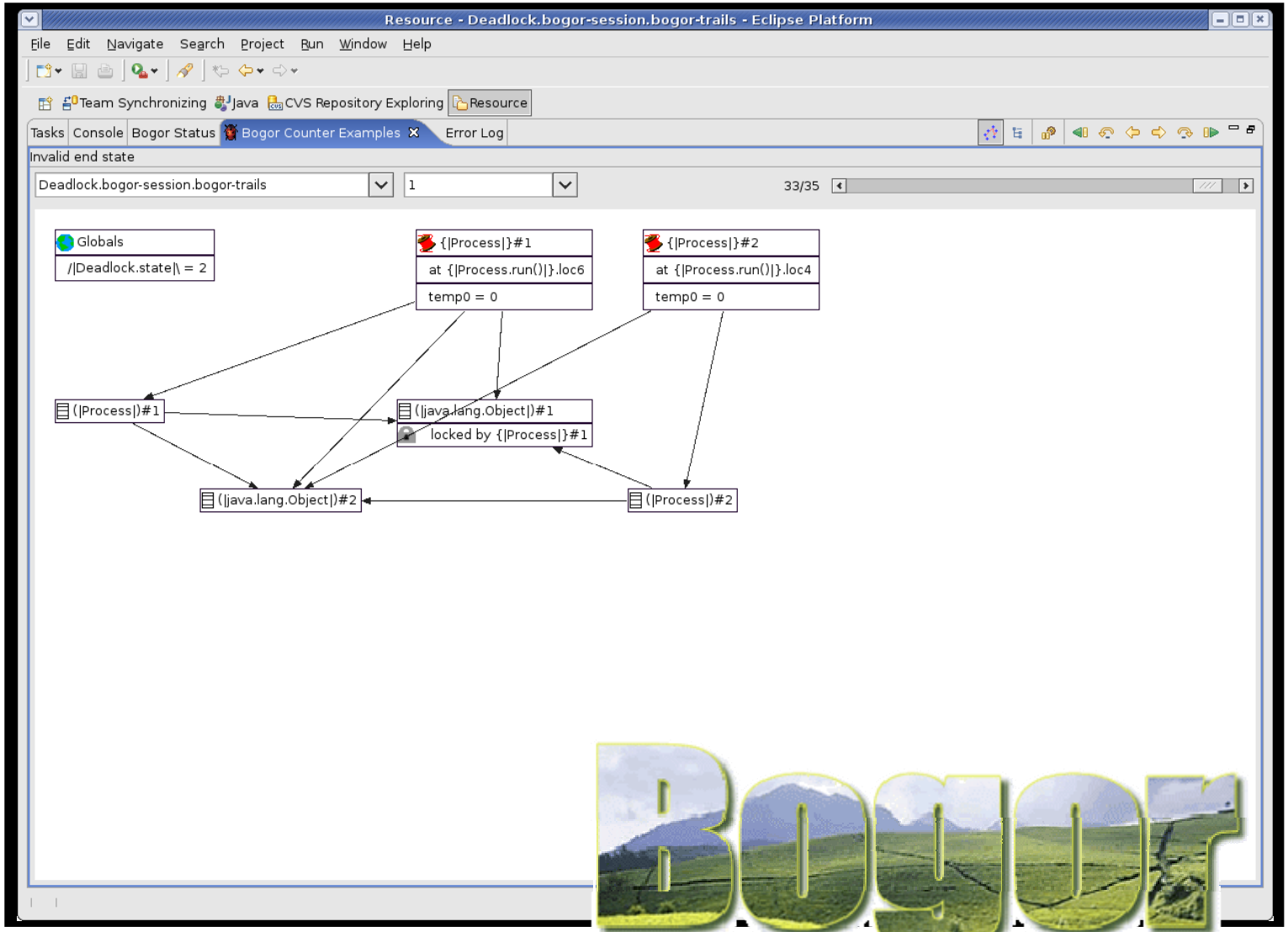
Lecture Organization (Cont'd)

- Lecture III: Bogor Architecture
  - Bogor configuration & initialization
  - DFS in Bogor (extensibility)
    - state representation
    - DFS stack
    - seen state set
- Lecture IV: Bogor Customizations
  - language extension
  - customization domains
    - Cadena models (avionics mission control system)
    - Java (.Net)
      - contract-reasoning using symbolic execution

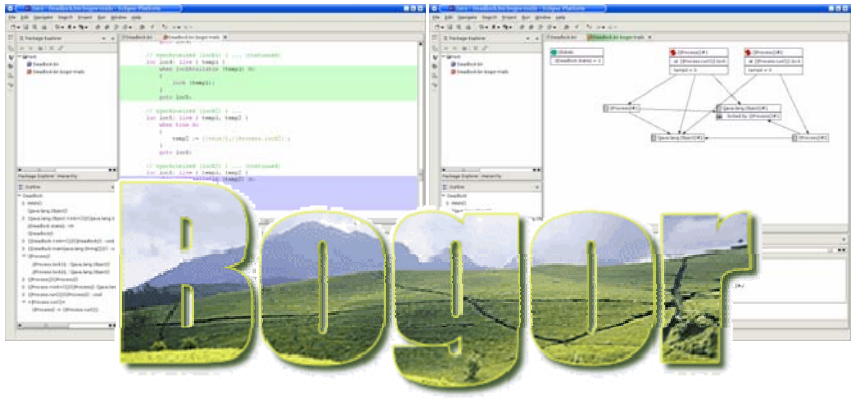
Bogor



Bogor – Software Model Checking Framework



Bogor – Direct support for OO software



Extensive support for checking concurrent OO software

Direct support for...

- *unbounded* dynamic creation of threads and objects
- automatic memory management (garbage collection)
- virtual methods, ...
- ..., exceptions, etc.
- *supports virtually all of Java*

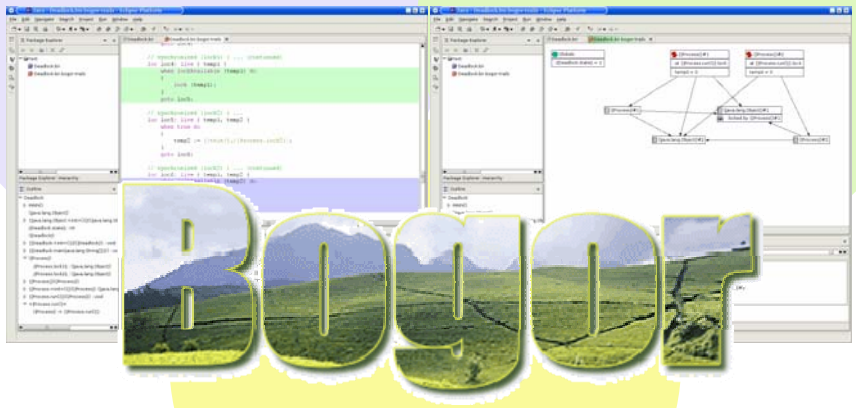
Software targeted algorithms...

- thread & heap symmetry
- compact state representation
- partial order reduction techniques driven by
  - object escape analysis
  - locking information

Bogor – Domain Specific Model-Checking

Modeling language and Algorithms
easily customized to different domains

Domain X



Domain Y

Domain Z

Extensible modeling language and *plug-in architecture* allows Bogor to be customized to a variety of application domains

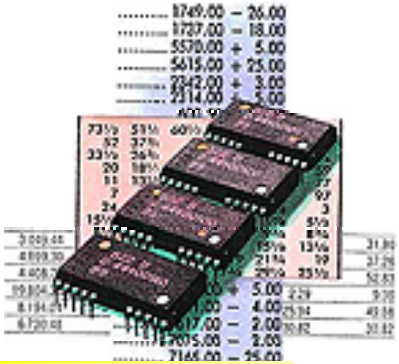
Build Your Own Model Checker!

Bogor Web Site

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

- Significant “out of the box” capabilities for supporting modern software systems
- Novel extension and customization facilities
  - used by us
  - used by others
- Documentation and support
- Pedagogical material and courseware

System Modeling Problem – Variety of Application Domains



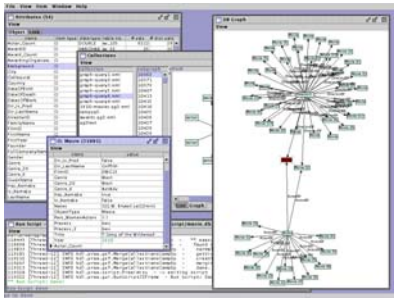
Hardware

Device Drivers

Avionics

Telephony

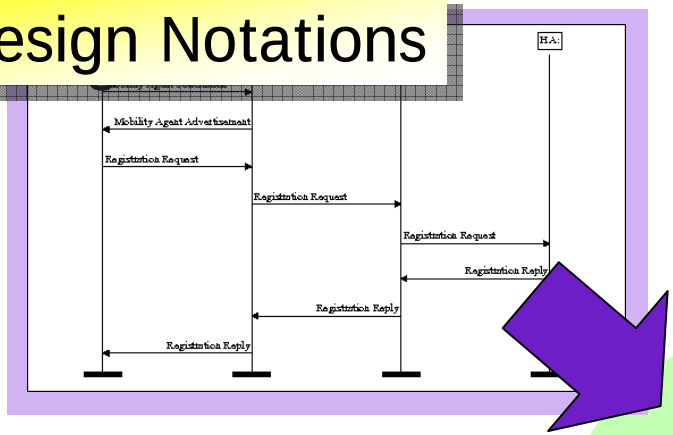
Automotive



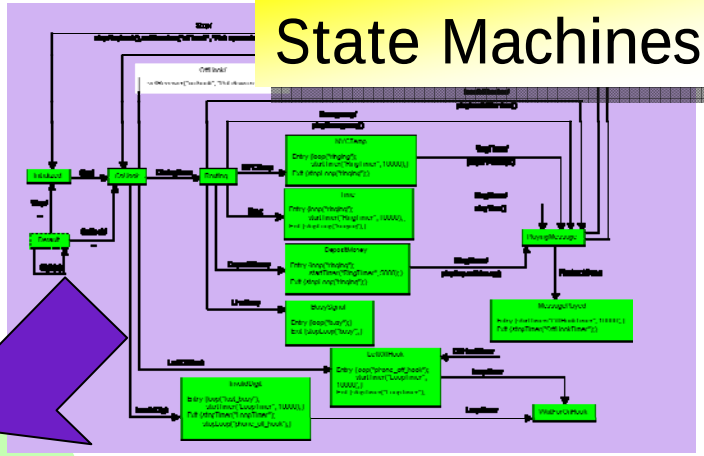
GUI

System Modeling Problem – Variety of System Descriptions

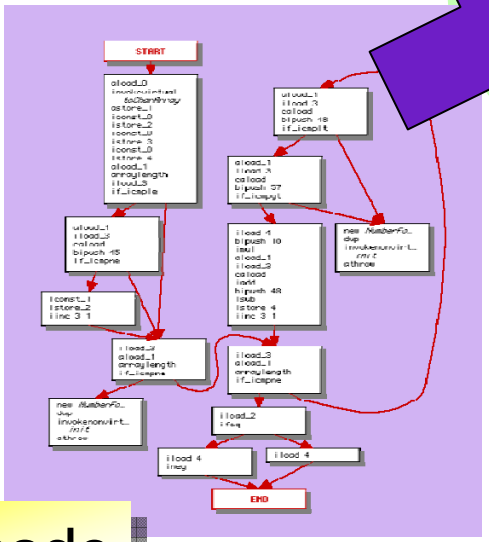
Design Notations



State Machines

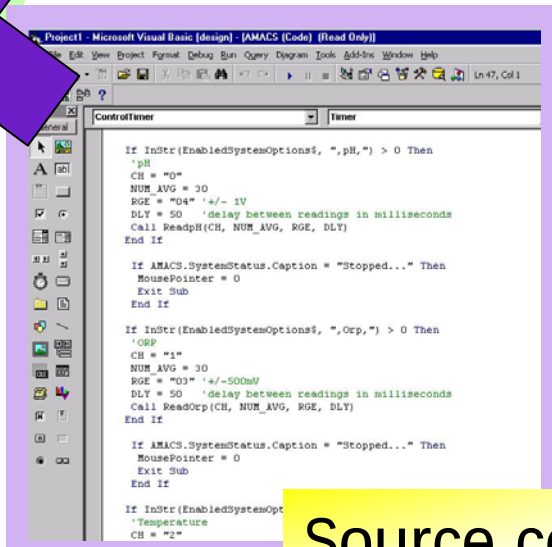


Model Checker



Byte code

Different levels of abstraction!

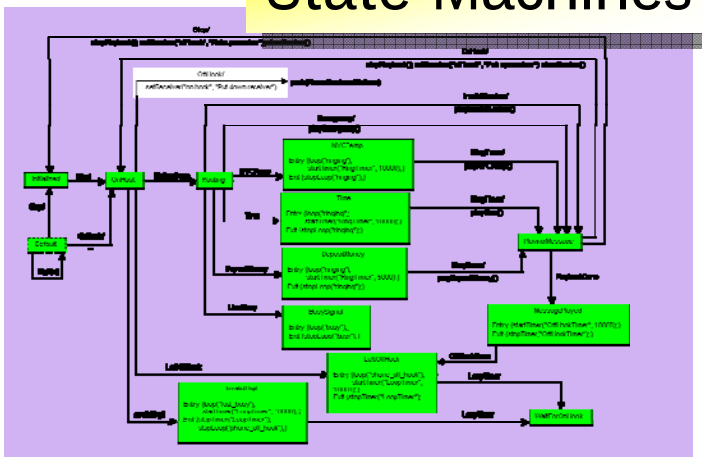


Source code

The Goal

Avionics

State Machines



Model-checking Engine

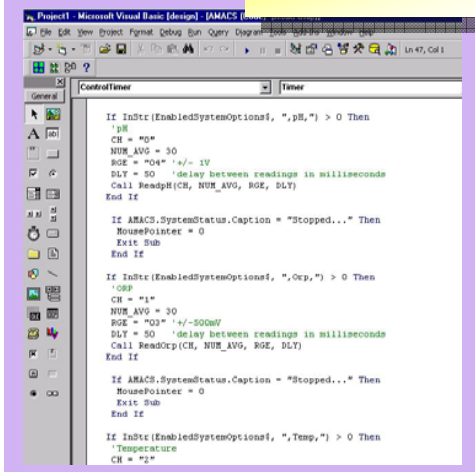
Domain & Abstraction Extensions

Abstract machine tailored to *domain* and *level of abstraction*

The Goal

Device Drivers

Source code



Model-checking Engine

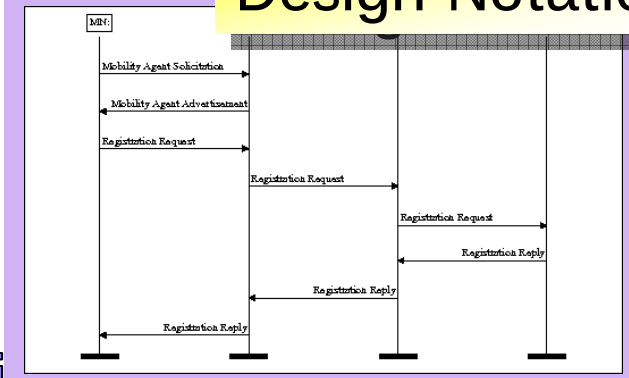
Domain & Abstraction Extensions

Abstract machine tailored to domain and level of abstraction

The Goal

Automotive

Design Notations



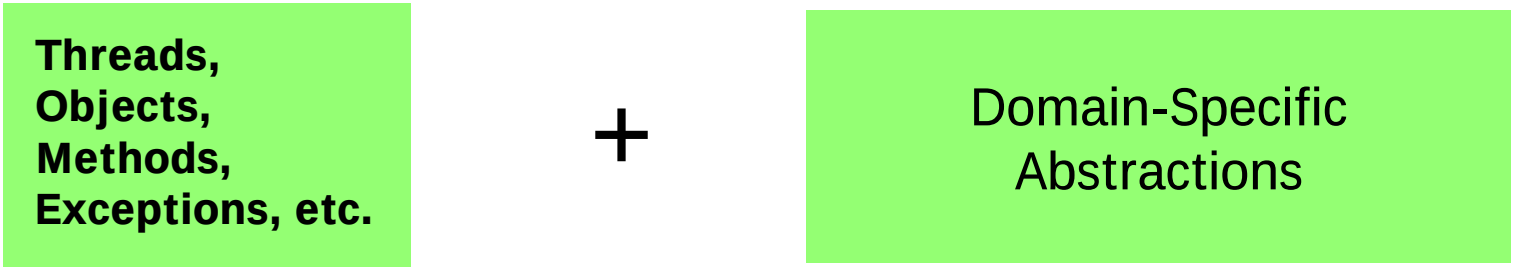
Model-checking Engine

Domain & Abstraction Extensions

Abstract machine tailored to domain and level of abstraction

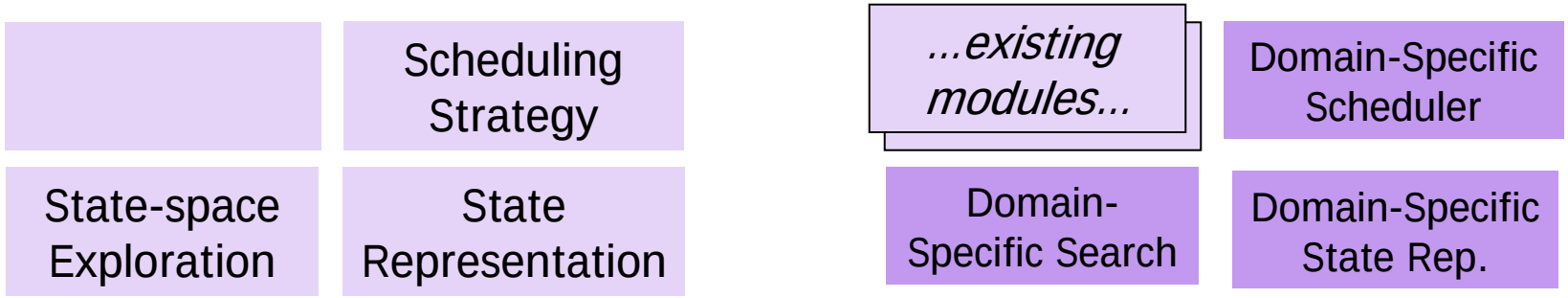
Customization Mechanisms

Bogor -- Extensible Modeling Language



Core Modeling Language

Bogor -- Customizable Checking Engine Modules



Core Checker Modules

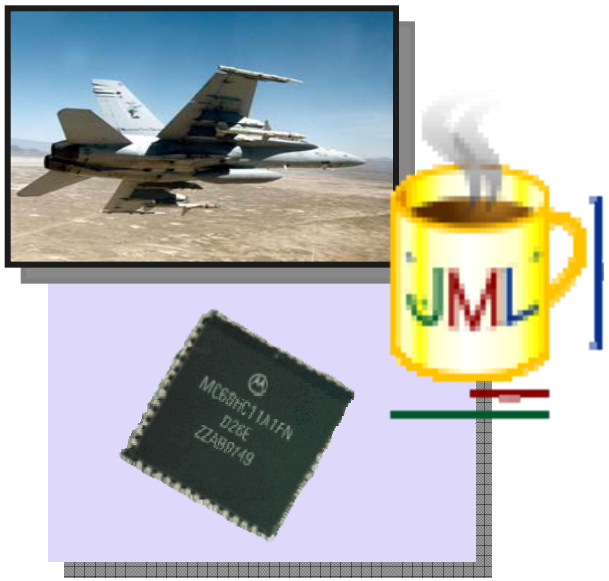
Customized Checker Modules

Domain Customizations

We are aware of more than 31 substantive extensions to Bogor that have been built by 19 people, only one of whom was the primary Bogor developer.

Examples

- Component-based avionics systems in *Cadena*
- Checking Java Modeling Language (JML) specifications
- Compositional Reasoning using Symbolic Execution
- Test Case Generation
- Multiagent System
- Modeling MPI (U Mass)
- Model checking Motorola M68HC11 machine code (BYU)



Outline

Overview

Bogor Modeling Language and UI

- Example: Dining philosophers
- Demo: Bogor UI in Eclipse

Modeling Java programs

- Java classes and methods
- Dynamic dispatch of methods
- Exceptions

Other BIR Features

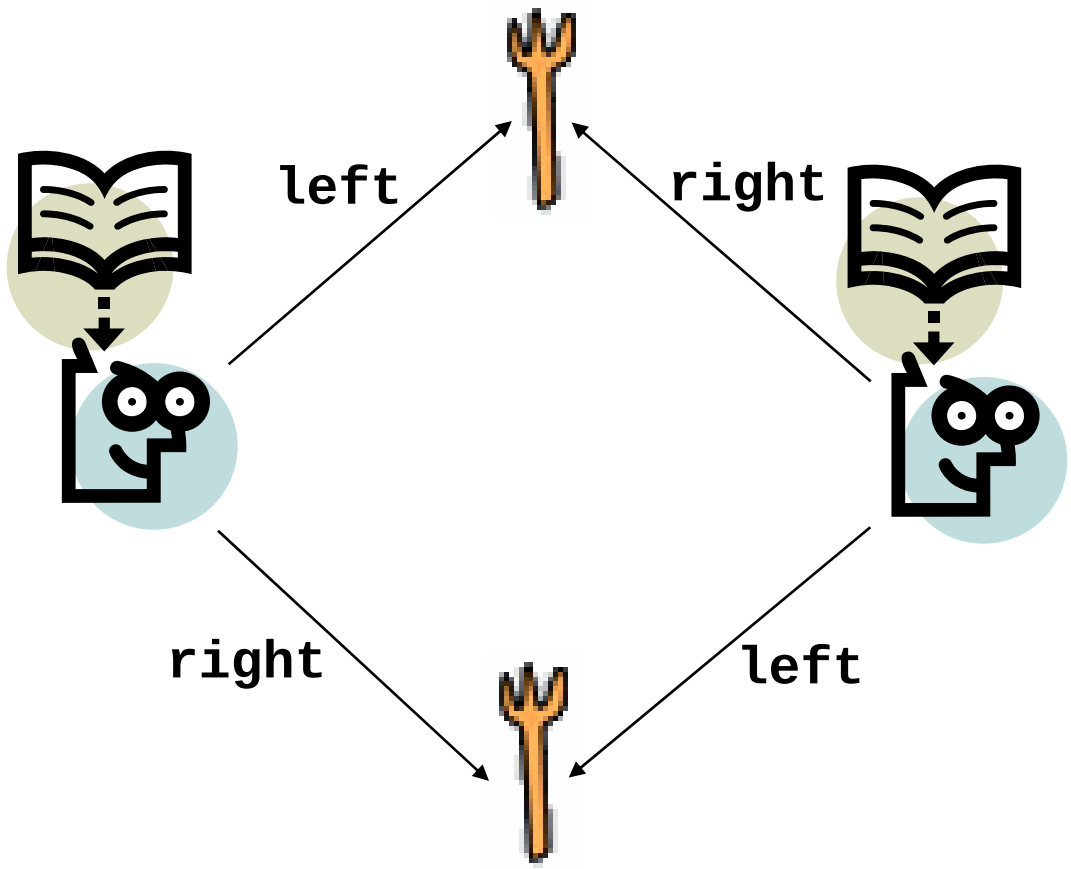
- High-level constructs
- Functional sub-language
- Example Extension: Channels

Bogor Modeling Language — BIR

BIR = Bandera Intermediate Representation

- Used as the intermediate language for the Bandera Tool Set for model-checking Java programs
- Guarded command language
  - `when` <condition> `do` <command>
- Native support for a variety of object-oriented language features
  - dynamically created objects and threads, exceptions, methods, inheritance, etc.

An Example — 2 Dining Philosophers



A BIR Example — 2 Dining Philosophers

```
system TwoDiningPhilosophers {
  record Fork { boolean isHeld; }

  main thread MAIN() {
    Fork fork1;
    Fork fork2;

    loc loc0:
      do {
        // create forks
        fork1 := new Fork;
        fork2 := new Fork;

        // start philosophers
        start Phil(fork1, fork2);
        start Phil(fork2, fork1);
      } return;
  }
```

```
 thread Phil(Fork left, Fork right) {
    loc loc0: // take left fork
      when !left.isHeld do {
        left.isHeld := true;
      } goto loc1;

    loc loc1: // take right fork
      when !right.isHeld do
      { right.isHeld := true; }
      goto loc2;

    loc loc2: // put right fork
      do { right.isHeld := false; }
      goto loc3;

    loc loc3: // put left fork
      do { left.isHeld := false; }
      goto loc0;
  }
}
```

A BIR Example — 2 Dining Philosophers

```
system TwoDiningPhilosophers {
  record Fork { boolean isHeld; }

  main thread MAIN() {
    Fork fork1;
    Fork fork2;

    loc loc0:
      do {
        // create forks
        fork1 := new Fork;
        fork2 := new Fork;

        // start philosophers
        start Phil(fork1, fork2);
        start Phil(fork2, fork1);
      } return;
  }
```

Uses a record to model forks

```
 thread Phil(Fork left, Fork right) {
    loc loc0: // take left fork
      when !left.isHeld do {
        left.isHeld := true;
      } goto loc1;

    loc loc1: // take right fork
      when !right.isHeld do
      { right.isHeld := true; }
      goto loc2;

    loc loc2: // put right fork
      do { right.isHeld := false; }
      goto loc3;

    loc loc3: // put left fork
      do { left.isHeld := false; }
      goto loc0;
  }
}
```

A BIR Example — 2 Dining Philosophers

```
system TwoDiningPhilosophers {
  record Fork { boolean isHeld; }

  ------------------------------------------------------------------------
  main thread MAIN() {
    Fork fork1;
    Fork fork2;

    loc loc0:
      do {
        // create forks
        fork1 := new Fork;
        fork2 := new Fork;

        // start philosophers
        start Phil(fork1, fork2);
        start Phil(fork2, fork1);
      } return;
  }
```

Thread declarations

```
  thread Phil(Fork left, Fork right) {
     loc loc0: // take left fork
       when !left.isHeld do {
         left.isHeld := true;
       } goto loc1;

     loc loc1: // take right fork
       when !right.isHeld do
       { right.isHeld := true; }
       goto loc2;

     loc loc2: // put right fork
       do { right.isHeld := false; }
       goto loc3;

     loc loc3: // put left fork
       do { left.isHeld := false; }
       goto loc0;
   }
}
```

A BIR Example — 2 Dining Philosophers

```
system TwoDiningPhilosophers {
  record Fork { boolean isHeld; }

  main thread MAIN() {
    Fork fork1;
    Fork fork2;

    loc loc0:
      do {
        // create forks
        fork1 := new Fork;
        fork2 := new Fork;

        // start philosophers
        start Phil(fork1, fork2);
        start Phil(fork2, fork1);
      } return;
  }
```

Local variable declarations

```
  thread Phil(Fork left, Fork right) {
                                  t fork
                                  o {
          left.isHeld := true;
        } goto loc1;

      loc loc1: // take right fork
        when !right.isHeld do
        { right.isHeld := true; }
        goto loc2;

      loc loc2: // put right fork
        do { right.isHeld := false; }
        goto loc3;

      loc loc3: // put left fork
        do { left.isHeld := false; }
        goto loc0;
    }
  }
```

A BIR Example — 2 Dining Philosophers

```
system TwoDiningPhilosophers {
  record Fork { boolean isHeld; }

  main thread MAIN() {
    Fork fork1;
    Fork fork2;

    loc loc0:
      do {
        // create forks
        fork2 := new Fork();

        // start philosophers
        start Phil(fork1, fork2);
        start Phil(fork2, fork1);
      } return;
  }
```

Control locations

```
thread Phil(Fork left, Fork right) {
    loc loc0: // take left fork
      when !left.isHeld do {
        left.isHeld := true;
      } goto loc1;

    loc loc1: // take right fork
      when !right.isHeld do
      { right.isHeld := true; }
      goto loc2;

    loc loc2: // put right fork
      do { right.isHeld := false; }
      goto loc3;

    loc loc3: // put left fork
      do { left.isHeld := false; }
      goto loc0;
  }
}
```

A BIR Example — 2 Dining Philosophers

Guarded transformations

…aka "guarded transitions", "guarded commands"

```
thread Phil(Fork left, Fork right) {
   loc loc0: // take left fork
     when !left.isHeld do {
        left.isHeld := true;
     } goto loc1;

   loc loc1: // take right fork
     when !right.isHeld do
     { right.isHeld := true; }
     goto loc2;

   loc loc2: // put right fork
     do { right.isHeld := false; }
     goto loc3;

   loc loc3: // put left fork
     do { left.isHeld := false; }
     goto loc0;
  }
}
```

When condition is true

Execute these statement(s) atomically

Trivially true guards

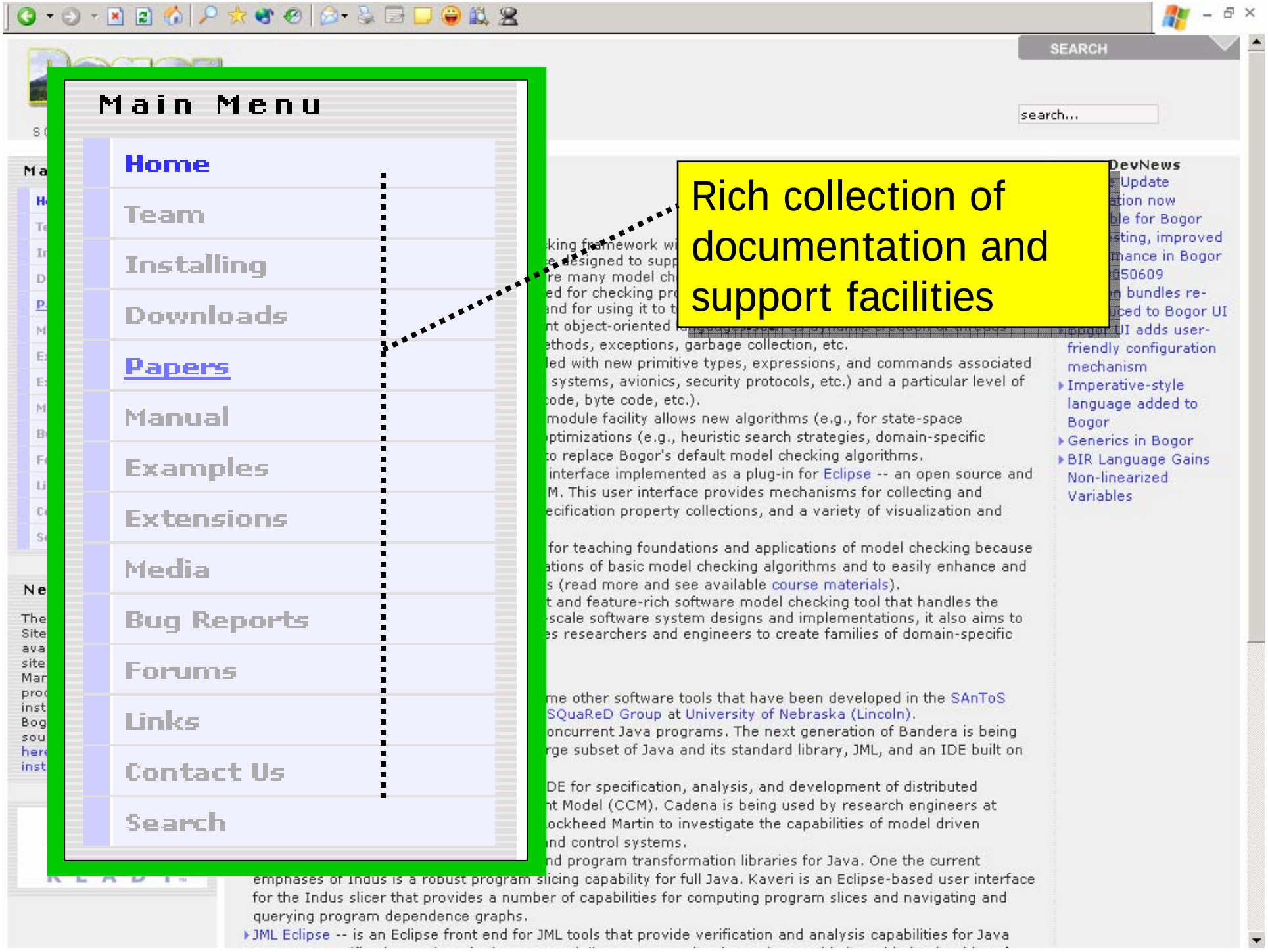
A BIR Example — 2 Dining Philosophers

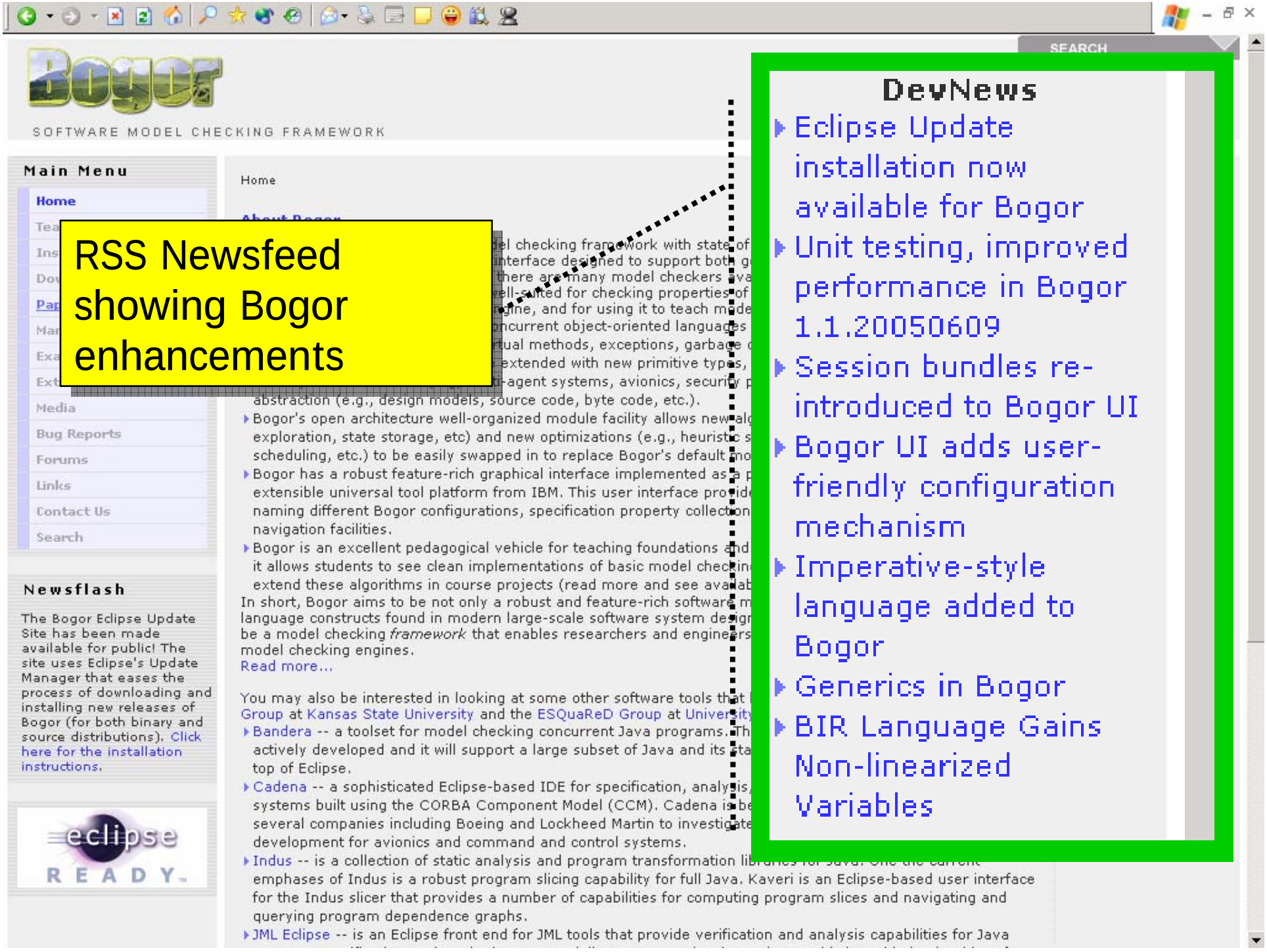
Demo

- Bogor BIR Editor
  - syntax highlighting
  - well-formed-ness checker
- Bogor Counter-example Display
  - states and transitions navigation
  - heap visualization
- BIR Session Wizard
  - creating new sessions
  - configuring Bogor

Bogor Online Resources

- Project website
  http://bogor.projects.cis.ksu.edu/
- Bogor User Manual
  http://bogor.projects.cis.ksu.edu/manual
- Distribution & Licensing
  - Freely downloadable from the Bogor project website
    - registration is required
  - Cannot redistribute the Bogor package, but you can redistribute Bogor extensions


Main Menu
Home
Team
Installing
Downloads
Papers
Manual
Examples
Extensions
Media
Bug Reports
Forums
Links
Contact Us
Search
Rich collection of documentation and support facilities
SEARCH
search...
DevNews
Update
tion now
e for Bogor
sting, improved
nance in Bogor
50609
n bundles re-
ced to Bogor UI
UI adds user-friendly configuration mechanism
Imperative-style language added to Bogor
Generics in Bogor
BIR Language Gains Non-linearized Variables


SOFTWARE MODEL CHECKING FRAMEWORK
Main Menu
Home
Media
Bug Reports
Forums
Links
Contact Us
Search
Newsflash
The Bogor Eclipse Update Site has been made available for public! The site uses Eclipse's Update Manager that eases the process of downloading and installing new releases of Bogor (for both binary and source distributions). Click here for the installation instructions.
Home
Read more...
RSS Newsfeed showing Bogor enhancements
DevNews
Eclipse Update installation now available for Bogor
Unit testing, improved performance in Bogor 1.1.20050609
Session bundles re-introduced to Bogor UI
Bogor UI adds user-friendly configuration mechanism
Imperative-style language added to Bogor
Generics in Bogor
BIR Language Gains Non-linearized Variables

Outline

Overview

Bogor Modeling Language and UI

- Example: Dining philosophers
- Demo: Bogor UI in Eclipse

Modeling Java programs

- Java classes and methods
- Dynamic dispatch of methods
- Exceptions

Other BIR Features

- High-level constructs
- Functional sub-language
- Example Extension: Channels

Modeling Java Programs

Java classes and methods

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Additional BIR identifiers

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
```

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
|]) -> {|A.foo()|}
|]) -> {|A.foo()|}
|]) -> {|C.foo()|}
```

(|…|), **/|…|**, **\|…|/**, **{|…|}**, and **+|…|+** are all BIR identifiers used to avoid name clashes

Modeling Java Programs

Records for Java classes

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

records are used to model Java classes (inheritance)

```
record (|A|)
       extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Static fields

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

static fields are modeled as global variables

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Java methods as functions

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

Java methods are modeled as functions

```
record (|A|)
       extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Dynamic dispatch of methods

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

Virtual tables are used to resolve dynamic dispatch of methods

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Dynamic dispatch of methods

```
public class A {
  public static int N;

  public int x;
  protected int y;

  public A() {…}

  public void foo() {…}
}

class B extends A {}

class C extends B {
  public void foo() {…}
}
```

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Dynamic dispatch of methods

```
function bar((|A|) a) {
 loc loc0: // invokespecial
   invoke {|A.foo()|} (a)
   goto loc0;

 loc loc1: // invokevirtual
   invoke virtual
                +|A.foo()|+ (a)
   goto loc1;
}
```

BIR function invocations models Java method invocations (static, special, virtual, or interface)

```
record (|A|)
      extends (|java.lang.Object|)
  { int /|A.x|\; int /|A.y|\; }
int \|A.N|/;

function {|A.<init>()|}
         ((|A|) [|this|]) …

function {|A.foo()|} ((|A|) [|this|]) …

record (|B|) extends (|A|) {}
record (|C|) extends (|B|) {}

function {|C.foo()|} ((|C|) [|this|]) …

virtual +|A.foo()|+ {
  (|A|) -> {|A.foo()|}
  (|B|) -> {|A.foo()|}
  (|C|) -> {|C.foo()|}
}
```

Modeling Java Programs

Exceptions

```
public static void baz() {
  try {
    …
  } catch (Throwable t) {
    …
  }
}
```

Throwable record

```
throwable
   record (|java.lang.Throwable|)
   extends (|java.lang.Object|) {…}

function {|baz|}() {
  (|java.lang.Throwable|) [|t|];

  loc loc0: do { … } goto loc1;

  loc loc1: do { … } return;

  loc loc2: do { … } goto loc1;

 catch (|java.lang.Throwable|) [|t|]
    at loc0 goto loc2;
}
```

Catch tables are used to keep track try-catch regions (order of declarations)

Outline

Overview

- Modeling Java programs
  - Java classes and methods
  - Dynamic dispatch of methods
  - Exceptions

- Bogor Modeling Language and UI
  - Example: Dining philosophers
  - Demo: Bogor UI in Eclipse

- Other BIR Features
  - High-level constructs
  - Functional sub-language
  - Example Extension: Channels

High-level BIR

Motivation

- while low-level BIR is suitable as a target for automatic translation, it is too cumbersome to use for manual modeling
- provides high-level constructs similar to most modern programming languages such as loops, try-catch, etc.
  - high-level BIR constructs are translated to low-level BIR before model checking starts

High-level BIR: Example

```
throwable record NullPointerException { }

throwable record IllegalArgumentException {}

record ListNode { ListNode next; int data; }

function sumAllElementsOver5(ListNode head) returns int {
  int firstElementData; int total;

  firstElementData := head.data;

  while head != null do
    choose
      when <head.data > 5> do total := total + head.data;
      else do skip;
    end
    head := head.next;
  end

  return total;
}
```

Statement sequencing, while loop, choose block, atomic expression, etc.

High-level BIR: Example

```
NullPointerException npe;
IllegalArgumentException iae;

ListNode n;
int length;

try
  length := sumAllElementsOver5(
                createNode(5,
                    createNode(7, null)));

  length := sumAllElementsOver5(null);
catch (IllegalArgumentException iae)
  skip;
catch (NullPointerException npe)
  assert false;
end
```

try-catch block

BIR Functional Sub-language

Motivation

- wants to allow complex queries of states while guaranteeing purity
  - very useful for specification purposes

Syntax and semantics

- similar to other functional languages (SML, *etc.*)
- currently only support first-order function

BIR Functional Sub-language: Example

```
record ListNode {
  Node next;
  int data;
}

fun sortedList(ListNode n)
             returns boolean =
  let
    ListNode next = n.next
  in
    next == null ?
      true
    : (n.data <= next.data ?
         sortedList(next)
       : false);
```

A recursive function to determine whether a given list is sorted (in ascending order)

BIR Extensions

```
extension Channel for MyChannel {
  // declaration of abstract types
  typedef type<'a>;

  // declaration of abstract expresions
  expdef Channel.type<'a> create<'a>(int);
  expdef boolean isEmpty<'a>(Channel.type<'a>);
  expdef 'a getFirst<'a>(Channel.type<'a>);

  // declaration of abstract actions/commands
  actiondef send<'a>(Channel.type<'a>, 'a);
  actiondef removeFirst<'a>(Channel.type<'a>);
}
```

BIR allows introduction of new abstract types and operations

```
Channel.type<int> chan;
int x;
…
  chan := Channel.create<int>(5); // ch 5 slots
  …
  Channel.send<int>(chan, 0); // send 0
  …
  x := Channel.getFirst<int>(chan); // recv 1st
  Channel.removeFirst<int>(chan);
```

Sample usage

BIR Assessment

- Variety of application domains and system level descriptions often work at different level of abstractions
- BIR provides features commonly found in modern programming languages
  - Dynamic creation of objects and threads, automatic memory management, *etc.*
- BIR provides extension mechanism to bridge the gap between system descriptions and BIR
  - can be extended on-demand basis
  - minimize changes of Bogor components
    - parser/lexer, symbol table, AST, type system, *etc.*