Software Model Checking using Bogor -- a Modular and Extensible Model Checking Framework

John Hatcliff
(http://www.cis.ksu.edu/~hatcliff)
SAnToS Laboratory
Kansas State University

Acknowledgments: These lecture materials and associated tools have been developed jointly by a number of researchers in the SAnToS Laboratory at Kansas State University. In particular, the lecture slides are taken from a larger set of pedagogical materials that the lecturer has developed jointly with Dr. Matthew Dwyer and Dr. Robby. The author wishes to especially thank Matthew Hoosier and other SAnToS group members for assisting in a variety of ways in the preparation of this material.

This work is funded in part by by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR 0306607), by NASA Ames Research Center (NAG-02-1209), by Lockheed-Martin, Rockwell-Collins, Honeywell Advanced Technology Center, Intel Corporation, and by an IBM Eclipse Innovation Grant.
Software Model Checking using Bogor — a Modular and Extensible Model Checking Framework

(http://bogor.projects.cis.ksu.edu)
(http://www.cis.ksu.edu/~hatcliff/ESSCaSS04)

John Hatcliff
ESSCaSS 2004

1 Overview of Lectures

State-space exploration strategies such as model checking are emerging as a popular technology for reasoning about behavioral properties of a wide variety of software artifacts including requirements models, architectural descriptions, design, implementations, and process models. The computational costs of model checking are well-known, and a decade of intensive research on general techniques for reducing the complexity of model checking has made model checking tools much more efficient. Yet, past experience has shown that domain-specific information can often be leveraged to obtain state-space reductions that go beyond general purpose reductions by customizing existing model checker implementations or by building new model-checking engines dedicated to a particular domain. Unfortunately, these strategies limit the dissemination of model checking techniques across a number of domains since it is often infeasible for domain experts to build their own dedicated model checkers or to modify existing model checking engines.

To enable researchers to more easily tailor a model checking engine to a particular software-related domain, we have constructed an extensible and modular explicit-state software model checking framework called Bogor. For treating realistic designs and implementations in widely-used languages such as Java and C#, Bogor provides a rich base modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. For these built-in features, Bogor employs state-of-the-art reduction techniques such as collapse compression, heap symmetry, thread symmetry, and partial-order reductions that leverage common heap and locking structures. For tailoring to specific domains, Bogor provides (a) mechanisms for extending Bogor’s modeling language with new primitive types, commands, expressions, and APIs associated with a particular application, and (b) a well-organized module facility for plugging customized domain-tailored components into the model-checking engine. Moreover, Bogor is designed to be easily encapsulated within larger domain-specific development and verification environments.

The goals of this series of lectures are to provide students with an overview of the foundations of explicit-state model checking as implemented by Bogor, and then to illustrate how Bogor can be customized to a particular domain.

Lecture I: Bogor Overview and Foundations of Explicit-State Model-checking

This lecture begins with an overview of the motivation for Bogor and gives a short demonstration of Bogor’s input language and user interface. This is followed by an introduction to the basic concepts of explicit-state model-checking. Emphasis is placed on the primary data structures involved in the model-checker’s depth-first search, as these are the data structures that one deals with when extending Bogor’s input language or customizing internal modules of Bogor.

Lecture II: Writing Bogor Extensions – A Tutorial

A simple set example is used to illustrate how Bogor’s input language can be extended with new types, new expressions, and new commands. Also illustrated are the application programming interfaces (APIs) for the primary data structures of Bogor’s state-space exploration engine.

Lecture III: Checking Java Programs and JML Specifications

This lecture will describe how Bogor supports model-checking of Java programs and rich specifications written in the Java Modeling Language (JML). This gives insight into how the sophisticated features of a high-level language such as methods, inheritance, exceptions, etc. can be supported in Bogor, and how Bogor’s reduction strategies (e.g., partial-order reductions) are tailored to features specific of the state of object-oriented software execution such as heap and locking structures. If time permits, this lecture will also briefly survey the Bandera Temporal Specification Patterns system that a number of researchers and practitioners have found useful for constructing temporal logic specifications.

Lecture IV: Designing Component-based Systems in Cadena and Checking Cadena Designs in Bogor

This lecture gives an overview of Cadena—an integrated development environment that we have built for the design, analysis, and implementation of systems built using the CORBA Component Model. Cadena is currently being used by research engineers at several companies including Boeing and Lockheed-Martin to demonstrate the effectiveness of model-driven component-based product-line development for avionics and command-and-control systems. Cadena uses Bogor to model-check high-level system designs, and this lecture describes how Bogor’s input language and underlying state-space exploration algorithms were customized to directly support the CORBA real-time event channel middleware infrastructure used in mission- and command-control applications.

At http://www.cis.ksu.edu/~hatcliff/ESSCaSS04 students can find materials specifically associated with these lectures including:

(a) the distribution of Bogor which includes user manual, tutorials, Bogor source code, and API documentation, and
(b) electronic versions of lecture slides, supplementary lecture notes, guided exercises, and associated research papers for background reading.

These lectures report on work carried out jointly with Dr. Matthew Dwyer, Dr. Robby, and other SAnToS group members.
2 Overview of SAnToS Tools

Researchers at Laboratory for Static Analysis and Transformation of Software (SAnToS Laboratory) at Kansas State University have created several tools for design, analysis, and verification of software. Below we give a brief overview of these tools.

2.1 Bandera—A Tool Set for Verification of Java Programs

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin, JPF, and JPP). Both program slicing and user extensible abstract interpretation components are applied to customize the program model to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

For the next generation of Bandera (planned for release at the end of 2004), the entire Bandera code-base has been completely redesigned and rewritten and implemented in IBM's Eclipse open-source IDE. Moreover, the back-end translations to Spin, JPF, etc. have been replaced by a dedicated translation into Bogor. These changes have greatly increased the robustness, scalability, and usability of Bandera.

Project URL: http://bandera.projects.cis.ksu.edu

2.2 Bogor—An Extensible and Modular Software Model Checker

Bogor is a highly customizable and modular model checking framework aimed at easing the development of robust and efficient domain-specific model checkers for verification of dynamic and concurrent software. It provides a rich and extensible modeling language including features that allow for dynamic creation of objects and threads, garbage collection, dynamic dispatch of methods, and exception handling.

The extensible modeling language allows user-defined abstract data types and abstract operations as first class constructs. This is particularly useful when customizing Bogor to a particular family of software artifacts. Furthermore, its open modular design eases the task of customization to accommodate, for example, variations in scheduling policies, search modes for state exploration, state encodings, and checkers for specification languages.

Bogor employs state-of-the-art reduction techniques such as collapse compression, heap symmetry, thread symmetry, and partial-order reductions. Bogor has been successfully customized for efficient verification of realistic Java programs in the Bandera project and real-time avionic systems in the Cadena project.

We believe that Bogor can be especially useful to researchers interested in experimenting with new modeling languages features and new model checking algorithms.

2.3 Cadena—A Development Environment for Design, Analysis, of Verification of Component-based Systems

The use of component models such as Enterprise Java Beans and the CORBA Component Model (CCM) in application development is expanding rapidly. Even in real-time safety/mission-critical domains, component-based development is beginning to take hold as a mechanism for incorporating non-functional aspects such as real-time, quality-of-service, and distribution.

To form an effective basis for the development of such systems, we have built Cadena—an integrated environment for building and modeling CCM systems. Cadena provides the following capabilities:

- A collection of light-weight specification forms that can be attached to CCM’s component Interface Definition Language (IDL) to specify mode variable domains, intra-component dependencies, and component state-transition semantics. These forms have a natural refinement order so that useful feedback can be obtained with little annotation effort, and increasing the precision of annotation yields more precise analysis. In addition, Cadena specifications allow developers to specify the same information in different ways, achieving a form of checkable redundancy that is useful for exposing design flaws.

- Dependency analysis capabilities allow tracing inter/intra-component event and data dependencies, as well as algorithms for synthesizing dependency-based real-time and distribution aspect information.

- A component assembly framework supporting a variety of visualization and programming tools for developing component connections.

- Integration with both C++ and Java CCM implementations including CIAO (a C++ implementation built on top of the ACE/TAO real-time CORBA implementation) and OpenCCM (a Java implementation that runs on top of a number of open source Java ORBs).

- A novel model-checking infrastructure dedicated to event-based inter-component communication via real-time middleware enables system design models (derived from CCM IDL, component-assembly descriptions and annotations) to be model-checked for global system properties.

- Java component stub and skeleton code generated using the CCM IDL compilers of OpenCCM or CIAO.

The Cadena tools are implemented as plug-ins to IBM’s Eclipse IDE. This provides an end-to-end integrated development environment for CCM-based Java systems moving from editing of component definitions and connections information to editing and debugging of auto-generated code templates.

Cadena is currently being used by research engineers at several companies including Boeing and Lockheed-Martin to demonstrate the effectiveness of model-driven component-based product-line development for avionics and command-and-control systems. We are actively collaborating with researchers at the University of Vanderbilt (developers of CIAO) to more effectively support model-driven development of distributed real-time embedded systems.

Project URL: http://cadena.projects.cis.ksu.edu

2.4 Indus—A Toolkit to Customize and Adapt Java Programs

Indus is a toolkit of program analyses and transformations targeted towards customization and adaptation of Java programs. In terms of software, Indus provides a library with the following modules/features (with more coming) that the users can combine to realize customized analyses, transformations, and tools.

- A generic and primitive data flow analysis framework targeted towards inter-procedural analyses.

- An object-flow/points-to analysis that built from the above framework and that is used to construct precise call-graphs and thread-graphs.
• An assortment of program analyses.
  – monitor analysis,
  – escape analysis,
  – safe lock analysis, and
  – dependence analyses - control, sequential intra/inter-procedural data, divergence, synchronization, interference, and ready.
• A customizable program slicer for Java.
• A framework to glue various modules of the library to realize dedicated implementations.
• A collection of useful data structures and algorithms that are often required during program analysis/transformation.

In terms of tools, Indus provides a customizable implementation of the program slicer that generates executable and non-executable variants of backward, forward, and complete slices. Upcoming versions of Indus will include a partial evaluation/specialization engine for Java.

Kaveri, is a subproject of Indus that delivers the above slicer as a plugin in Eclipse. At present, the user can slice Java projects and view the slice in a Java editor in Eclipse. Future versions of the plugin will provide intuitive ways to understand the slice by chasing dependence and querying the slice.

Project URL: http://indus.projects.cis.ksu.edu

3 Overview of SAnToS Educational Material

3.1 Software Specifications
This course emphasizes tool-based formal specification methods and their role in the design and implementation of software systems. We aim to cover methods that span the development process ranging from high-level semantic modeling (Alloy), to system architecture design (USE), to coding and debugging (JML and ESC/Java).

Alloy (developed by Daniel Jackson and his students at MIT), in particular, “brings specifications to life” by allowing designers to query constructed models via its constraint analyzer and to easily assess the impact of model refinements. In addition, Alloy has a readable graphical and textual notation and it has a very small and semantically clean core language. We believe that all these factors make it an excellent pedagogical vehicle and a tool that can be applied to realistic problems with a fairly large return on the analyst’s effort. (Note: our materials use an earlier version of Alloy (Alpha 1.1) rather than the most recent version, because when we started this version of the course in January 2002, the documentation for the older version was more complete—January 2005 the course will transition to the latest version of Alloy).

We cover aspects of UML due to its current popularity. The USE tool (developed by Martin Gogolla and Mark Richters from the University of Bremen) is very robust and provides some very useful capabilities for reasoning about OCL enriched UML specifications. We believe that students may well use this tool in their design/development work in the future.

We include a module on ESC/Java (developed by researchers at Compaq/SRC) because we want to cover discussion of code-level specifications. The fact that ESC/Java provides some support for automatically checking these specs is in line with our preference for tool-supported methods. In Spring 2005, the course will also include checking specifications written in the Java Modeling Language (JML) (developed by Gary Leavens and colleagues) using Bogor.

The course distribution for instructors includes a variety of pedagogical materials such as typeset lecture notes and guided exercises, PowerPoint lecture slides, streaming video for our lectures, source code for lecture examples, weekly quizzes and solutions, homeworks and solutions, exams and solutions. A separate distribution for students includes only the lecture slides and examples.

Course URL: http://software-specs.courses.projects.cis.ksu.edu

3.2 Foundations and Applications of Software Model Checking
Modern computing applications increasingly require concurrent/distributed software systems that are extremely reliable. Unfortunately, current software validation techniques, such as inspections and testing, are failing to provide high levels of assurance of correctness for these systems due to system size and complexity as well as the fundamental difficulties of reasoning about state/event sequences in concurrent behavior.

Model-checking techniques (now widely used for hardware verification) hold promise for establishing crucial behavioral properties of complex software because they can automatically check to see if an abstract finite-state transition system model of the software conforms to a given state/event sequence property. If the model fails to satisfy the property, the model-checker gives a counterexample—a path through the model’s transitions that violates the property. This can be used to locate and correct the corresponding software defect.

This course emphasizes a practical and project-oriented approach to learning the technical foundations of model checking and methodologies for applying model-checking tools to realistic systems. Foundational topics covered include basic explicit-state reachability algorithms, temporal specification formalisms including LTL and CTL, partial-order reductions, state-space representations (collaps compression, etc.), and alternate search strategies. In an approach similar to that used in compiler courses, these foundational and theoretical concepts are reinforced by having students implement key components of an explicit state model-checker.

The Bogor model checker developed at Kansas State University plays a central role in the course. Students learn to apply Bogor to model and analyze simple concurrent systems that illustrate basic concepts of state-space exploration. Programming projects involve (re)implementing or modifying the core modules of Bogor’s model checking engine, or implementing new modeling language primitives using Bogor’s extensible modeling language. In addition to simply reinforcing the central concepts of model checking, the overall goal of these implementation exercises is to move students to the point where they can effectively develop model-checking tools and associated methodologies for verification of real world systems by tailoring Bogor to different application domains.

Methodological aspects of model checking (and Bogor, in particular) are also emphasized. This includes repeatable strategies for capturing concurrent/distributed systems as effective verification models, applying abstraction and other state-space reducing model transformations, and using a pattern-based approach to constructing temporal specifications.

The course distribution for instructors includes a variety of pedagogical materials such as typeset lecture notes and guided exercises, PowerPoint lecture slides, streaming video for our lectures, source code for lecture examples, weekly quizzes and solutions, homeworks and solutions, exams and solutions. A separate distribution for students includes only the lecture slides and examples.

Course URL: http://model-checking.courses.projects.cis.ksu.edu
Contents

1 Foundations .................................. 5
  1.1 BIR ................................................. 6
  1.1.1 A simple BIR system .................... 6
  1.1.2 States and transitions ................... 7
  1.2 Interleavings, Schedules, and Computation Trees ........ 12
  1.3 Executing (Simulating) BIR Systems with Bogor .......... 16
  1.4 Verifying BIR Systems with Bogor ............... 19
Chapter 1

Foundations

This chapter presents basic concepts associated with explicit-state model-checking — focusing on the primary data structures required for implementing a simple depth-first state-space exploration algorithm.

Our reasons for taking this approach are as follows.

• The depth-first search algorithm that will be presented forms the foundation of the more sophisticated strategies used in many popular model-checkers, especially software-oriented model-checkers. Thus, material in subsequent chapters will explain these more sophisticated strategies by making modifications and extensions of the basic algorithm presented here.

• Many readers that are new to model-checking are unable to predict even for simple systems the state-space size and thus the overall costs of model-checking. By understanding the basic algorithms and data structures, readers will be able to understand how concurrent computation gives rise to interweavings, and the relative number of associated interweavings.

• As stated earlier, model-checking is often effective when adapted to particular domains. Thus, researchers interested in developing and applying model-checking to their particular domain of interest need to understand enough of the internal details to be able to adapt confidently an existing model-checker.

Our linguistic vehicle for introducing model-checking concepts is a simple notation for describing concurrent systems called BIR. In its full form, BIR is a sophisticated language called the Bandera Intermediate Representation (BIR) that is used in the Bandera Tool Set [?] and the Bogor model-checker to represent models of concurrent object-oriented systems. These models may be coded directly in BIR or constructed automatically by translations from Java source code, Java byte code, or higher-level design notations. Because we will not need the full expressive power of BIR, we will restrict our initial system descriptions to a very simple subset of BIR.

The simple model of concurrent systems represented by BIR is rich enough to illustrate wide body of concepts associated with model-checking including state-space exploration, assertion and deadlock-checking, LTL checking, various forms of search optimization including partial-order reductions, etc. The simple structure of the BIR constructs we will use allows us to provide a clean presentation of basic issues, without being distracted by the more complicated features of full BIR. In later chapters, we will add more complicated linguistic features to BIR including dynamically created objects, references, functions/methods, etc. which are all useful for modeling modern software systems.

We begin by introducing the syntax and semantics of BIR (formal definitions can be found in appendices). We then present the notion of computation trees as a means of discussing and reasoning about the exploration of a concurrent system’s various schedules and execution traces. This will lead us into a discussion of Bogor, and how it can simulate a concurrent system by exploring a path found in a system’s computation tree, and how it can verify properties of a concurrent system by exploring all the paths in a system’s computation tree.

We will describe in detail simple representations for the main data structures needed to implement this exploration, and outline the depth-first exploration algorithm. We then present the first extensions to the algorithm which will implement dead-lock checking and a simple depth-bounded search. Finally, we conclude this chapter by applying Bogor to a number of simple concurrent systems.

1.1 BIR

1.1.1 A simple BIR system

Figure 1.1 presents the BIR code for a version of the classical dining philosophers system which we used in Chapter ?? to motivate issues in concurrent systems. We use this code to introduce the features of BIR.

Typically, the data declaration section will describe a bounded data space by bounding basic data types (e.g., integer values are bounded by subranges). BIR’s primitive types include boolean, integer subranges, and enumerated types. In the example of Figure 1.1, the declarations of the two boolean variables fork1 and fork2 for the data declaration section.

Thread declarations are used to define independently executing transition systems that correspond to processes or threads of control. In Figure 1.1, the two thread declarations represent the independently executing actions associated with two different philosophers. Placing the qualifier active on a thread declaration indicates that the thread is to begin its execution immediately. If
the active qualifier is omitted from a thread declaration, the thread must be started explicitly by another thread using the BIR construct start.

A thread body consists of a sequence of locations. When system execution begins, control in each thread begins at the start location – the first location in the thread’s location sequence. At any given time, a thread is at one of its locations, called the current location of the thread. The location of a thread is an abstraction of the program counter that would mark the next machine-instruction to be executed in a lower-level representation of the thread, and we often refer to a thread’s “current location” as its program counter.

Each location is the source of one or more guarded transformations. Each transformation consists of a boolean guard expression followed by a sequence of actions ending in the target location indicating the source of the next transformation in the thread to be executed. The intuition is that each time a thread is scheduled to make a step in the system’s computation, the underlying scheduler selects for execution a transformation associated with the thread’s that has a guard expression that evaluates to true. The chosen transformation’s actions are executed in sequence, and the system’s data variables are updated (atomically) to produce the next execution state.

1.1.2 States and transitions

Let us now take this intuition about thread actions, and develop a more formal description of a system’s execution. An execution of a BIR system can be viewed as a sequence of atomic steps or transitions between system states. Intuitively, a BIR state holds all information that is necessary to carry on computation starting from a particular point in the system’s execution – that state holds current location for each thread, along with the values of all the variables from the system’s data declaration section.

Figure 1.2 presents the states and transitions of the beginning of one possible execution of the BIR system of Figure 1.1. The initial state of the system shows the initial values of the program counters for both threads and the initial values of the system’s variables. The notation $pc_1 := 0$ represents the fact that the program counter for the Philosopher1 thread is set to the initial location loc0. The notation $\text{true}$ represents the fact that thread 1 has executed a transition out of location 0. Thus, the fourth transition in the trace of Figure 1.2 corresponds to Philosopher1 executing its transition leading out of location loc2.

Given this intuition for states and transitions, we now use the notion of state transition system $\Sigma_{BIR}$ to define formally the behavior of a BIR system. A state transition system $\Sigma$ is a quadruple $(S, T, S_0, L)$ with a set of states $S$, a set of transitions $T$ such that for each $a \in T$, $\alpha \subseteq S \times S$, a set of initial states $S_0$, and a labeling function $L$ that maps a state to a set of primitive propositions that are true at s (we will not use the labeling function until Chapter 7 where property languages are addressed).

In the state transition system $\Sigma_{BIR}$ for Figure 1.1, there are 64 states in $S$ – one for each possible combination of thread locations (4 different locations for each thread, yielding 16 different pairs of locations) and variable values (2 different values for each boolean variable, yielding 4 different pairs). The initial state of the system $S_0$ has all threads in their initial locations and all variables have the default value of their type (the default value of boolean variables is "false"). Note that not all of the states of $S$ are reachable in $\Sigma_{BIR}$. For example, it is impossible to start from the initial state and reach a state such as $[pc_1 := 2, pc_2 := 0, fork_1 := "false", fork_2 := "false"]$. 

Figure 1.1: A BIR system modeling 2-dining-philosophers

```
1.1. BIR

system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;
  active thread Philosopher1() {
    loc0: // take first fork
    when (fork1 do { fork1 == true })
    goto loc1:
  }
  active thread Philosopher2() {
    loc0: // take second fork
    when (fork2 do { fork2 == true })
    goto loc1:
  }
}
```

Figure 1.2: States and transitions for a 2DiningPhilosophers execution
where thread Philosopher1 is at location loc2 and where neither fork is held.

To save space, as we continue our discussion of states in the dining philosopher example, we will abbreviate the state notation by dropping the names of state variables. Thus, the state immediately above will be denoted as follows:

\[
[2.0, \text{false}, \text{false}]
\]

For an arbitrary BIR system, in the corresponding state transition system there will be a transition \( \alpha \) in \( T \) for each guarded transformation in the BIR code. For example, in \( \Sigma_2DP \), there are exactly eight transitions in \( T \) – one for each of the four transformations in each of the two threads.

A transition can be thought of as a state transformer that, given a state as input, produces one or more states as output. A transition is deterministic if for every state \( s \) there is at most one state \( s' \) such that \( \alpha(s, s') \). When \( \alpha \) is deterministic (i.e., when \( \alpha \) can be view as a function from states to states instead of the more general relation between states as introduced in the definition of transition systems above), we write \( s' = \alpha(s) \) instead of \( \alpha(s, s') \). In our technical discussions, we hardly ever consider non-deterministic transitions, and special notice will be given when we do so. Note that all the transitions of the dining philosopher system of Figure 1.1 are deterministic.

For some more intuition on transitions, let \( \alpha_{11} \) represent the transition associated with location loc1 in Philosopher1, and consider the functional behavior of \( \alpha_{11} \) on some example states:

\[
\alpha_{11}(1.0, \text{true}, \text{false}) = [2.0, \text{true}, \text{true}]
\]

Note that the state supplied as the argument in the second application above is actually unreachable, but that does not affect the definition of \( \alpha_{11} \). A more interesting situation is to consider is when \( \alpha_{11} \) is undefined – that is, when it does not produce an output for the supplied argument. Consider the following states (\( \alpha_{11} \) is undefined on all of them):

\[
[1.1, \text{true}, \text{false}]
\]

\[
[0.0, \text{false}, \text{false}]
\]

\[
[1.2, \text{false}, \text{true}]
\]

\( \alpha_{11} \) is undefined on the first state because the guard of \( \alpha_{11} \) requires that for\( \text{loc2} \) be not held ("false") before the transition can fire. \( \alpha_{11} \) is undefined on the second state because their is an implicit guard associated with the transition that requires Philosopher1’s program counter to be at location loc1 before the transition can fire. Even though the third state above is unreachable in the system, \( \alpha_{11} \) is undefined on it for reasons similar to the first state.

In general, for a transition \( \alpha \in T \), we say that \( \alpha \) is enabled in a state \( s \) if it is defined on \( s \), i.e., there is a state \( s' \) such that \( \alpha(s, s') \) holds. Otherwise, \( \alpha \) is disabled in \( s \). The set of transitions enabled in \( s \) is \( \text{enabled}(s) \), and the set of transitions enabled in \( s \) belonging to thread \( t \) is \( \text{enabled}(s, t) \).

Here are some examples.

\[
\begin{align*}
\text{enabled}[0, 0, \text{false}, \text{false}] = \{ &\alpha_{11}, \alpha_{22}\} \\
\text{enabled}[1, 0, \text{true}, \text{false}] = \{ &\alpha_{11}, \alpha_{22}\} \\
\text{enabled}[2, 0, \text{true}, \text{true}] = \{ &\alpha_{11}\} \\
\text{enabled}[1, 1, \text{true}, \text{true}] = \emptyset \\
\text{enabled}[1, 0, \text{true}, \text{false}, 1] = \{ &\alpha_{11}\} \\
\text{enabled}[2, 0, \text{true}, \text{true}, 2] = \emptyset
\end{align*}
\]

We denote the program counter of a thread \( t \) in a state \( s \) by \( pc(s, t) \). We write \( \text{current}(s, t) \) for the set of transitions associated the current control point \( pc(s, t) \) of thread \( t \) (this set will include \( \text{enabled}(s, t) \) as well as any transitions of \( t \) at \( pc(s, t) \) that may be disabled). Also, \( \text{current}(s) \) represents the union of current transitions at \( s \) for all active threads.

Here are some examples.

\[
\begin{align*}
\text{pc}(3, 0, \text{false}, \text{false}, 1) = 0 \\
\text{pc}(3, 2, \text{true}, \text{true}, 2) = 2 \\
\text{current}(2, 0, \text{true}, \text{true}) = \{ &\alpha_{12}, \alpha_{23}\}
\end{align*}
\]

Note that even though we will usually not consider non-deterministic transitions, this does not eliminate non-determinism in a thread – non-determinism is simply represented by multiple enabled transitions at a single control location in a thread. For example, the following BIR fragment shows a location with more than one transition leading out.

\[
\begin{align*}
&\text{loc loc1:} \\
&\text{loc loc2:}
\end{align*}
\]

In a state where control is at loc0 and both A and B are true, then both transitions are enabled, and is selected for execution non-deterministically.

For each transition \( \alpha \), we assume that we can determine, among other things, a unique identifier for a thread \( t \) that executes \( \alpha \) and the set of variables that are read or written by \( \alpha \).

A path \( \pi \) from a state \( s \) is a finite or infinite sequence such that \( \pi = s \leadsto s_1 \leadsto \ldots \leadsto s_n \leadsto \ldots \), such that \( s = s_0 \) and for every \( i, \alpha(s_i) = s_{i+1} \).

More details of the relationship between BIR systems and state transition systems can be found in Appendix 7.7.

**Exercises**

The goal of these exercises is to help the reader become acquainted with the basic aspects of BIR syntax and semantics.

1. Write a BIR program to compute the sum of the integers between 1 and 20 (inclusive).

2. (Guided) For the bounded buffer example of Figure 1.3, give three examples of a state \( s \) that is valid in the sense that the values for program
Figure 1.4: A BIR system modeling a simple bounded buffer

system BoundedBuffer {
    boolean full := false;
    boolean locked := false;
    active thread Producer() {
        loc0:
        when unlocked do {
            locked := true;
        } goto loc1;
        loc1:
        when full do {
            locked := false;
        } goto loc3;
        loc3:
        when full do {
            locked := false;
        } goto loc2;
        loc2:
        do {
            full := true;
        } goto loc3;
        loc0:
    }
}

active thread Consumer() {
    loc0:
    when locked do {
        locked := true;
    } goto loc1;
    loc1:
    when full do {
        locked := false;
    } goto loc0;
    loc0:
    do {
        full := false;
    } goto loc0;
    loc0:
}

Figure 1.3: A BIR system modeling a simple bounded buffer

system ReadersWriters {
    constant Param {
        NUM_READERS := 2;
        NUM_WRITERS := 1;
    }
    // Invariant:
    // (nr == 0 & nw == 0) \land nw \leq 1
    // Invariant:
    // (nr == 0 & nw == 0) \land nw \leq 1
    // Invariant:
    // (nr == 0 & nw == 0) \land nw \leq 1
    active[Param NUM_READERS] thread Reader(int index) {
        loc0:
        when nr == 0 do {
            nr := nr + 1;
        } goto loc1;
        loc1:
        do {
            nr := nr + 1;
        } goto loc0;
        loc0:
    }

    active[Param NUM_WRITERS] thread Writer(int index) {
        loc0:
        when nw == 0 do {
            nw := nw + 1;
        } goto loc1;
        loc1:
        do {
            nw := nw + 1;
        } goto loc0;
        loc0:
    }
}

Figure 1.4: A BIR system modeling a simple systems of readers and writers

In this section, we discussed the notion of an execution path as a sequence of states where each state is actually reachable from the initial state of the system but yet s is actually unreachable from the initial state of the bounded buffer program. Give a short explanation to justify your answer.

3. (GUIDED) In this section, given a transition α and a state s, we discussed that α could be either enabled or disabled in s.

   (a) For the bounded buffer example of Figure 1.3, give examples of three transitions and associated states such that the transitions are enabled in the associated states. Give a short explanation to justify your answer.
   (b) For the bounded buffer example of Figure 1.3, give examples of three transitions and associated states such that the transitions are disabled in the associated states. Give a short explanation to justify your answer.

4. (GUIDED) For the bounded buffer example of Figure 1.3, give an example of a state s where the set of current transitions of s is not the same as the set of enabled transitions of s. Give a short explanation to justify your answer.

5. (GUIDED) In this section, we discussed the notion of an execution path as a sequence of states where each state s_{k+1} was generated from its predecessor s_k by applying an transition α that is enabled in s_k. For the bounded buffer example of Figure 1.3,
   (a) give a sequence of six states that forms an execution for the system, and
   (b) give a sequence of six states that does not form an execution path for the system. Give a short explanation to justify your answer.

6. Carry out the instructions in Exercise 2 above for the readers/writers example of Figure 1.4.

7. Carry out the instructions in Exercise 3 above for the readers/writers example of Figure 1.4.

8. Carry out the instructions in Exercise 5 above for the readers/writers example of Figure 1.4.

1.2 Interleavings, Schedules, and Computation Trees

Figure 1.5 presents the BIR code for a system that we will use to illustrate the concept of instruction interleaving associated with concurrent execution, and the power of state-space exploration techniques such as model-checking for automatically reasoning about the effects of interleavings.
1.2. INTERLEAVINGS, SCHEDULES, AND COMPUTATION TREES

```
const PARAM { N = 1; }
typedefs byte int wrap (0..255);
byte x :: 1;
byte t1;
byte t2;
active thread Thread1 () {
  loc loc0 :
  when x != (byte) 0 do {
    t1 := x;
    goto loc1;
  } goto loc0;
  loc loc1 :
  do {
    t2 := x;
  } goto loc2;
  loc loc2 :
  do {
    x := t1 + t2;
  } goto loc0;
}
active thread Thread2 () {
  loc loc0 :
  when x != (byte) 0 do {
    t1 := x;
    goto loc1;
  } goto loc0;
  loc loc1 :
  do {
    t2 := x;
  } goto loc2;
  loc loc2 :
  do {
    x := t1 + t2;
  } goto loc0;
}
active thread Thread0 () {
  loc loc0 :
  do {
    assert (x != (byte) PARAM.N);
  } return;
}
```

Figure 1.5: A BIR system for illustrating instruction interleavings.

Threads Thread1 and Thread2 simply copy the value of variable x to temporary variables t1 and t2, and then assign the sum of the temporaries back to x. The role of Thread0, on the other hand, is different. Notice that its sole instruction is an assertion that x does not have the value N. Because this assertion is located in its own (independently executing) thread, it can fire at any moment in the program’s execution. Thread0, then, acts as a sort of “monitor” to check whether a bad value of x ever occurs.

In a multi-threaded system running on a single processor, a scheduler that is part of the run-time system picks the thread that should take the next execution step. Thus, the scheduling strategy determines a particular schedule—an ordering of instruction executions.

Can the assertion in SumToN be violated? Answering this question requires one to reason about possible schedules. Figure 1.6 displays two schedules that cause the assertion to be violated when N is set to 2.

In this figure, we have represented a schedule as sequence of states with each annotated arrow \( t \rightarrow s \) representing a transition between two states caused by thread \( t \) executing the instruction at location \( s \) of the thread. For example, \( 1 \rightarrow 0 \) represents the execution of the instruction at loc1 of Thread2.

By this point, the reader has probably surmised that there are a number of different schedules that can cause the assertion of SumToN to be violated. It is helpful to visualize the possible schedules of a particular system in the form of a computation tree consisting of system states as tree nodes and transitions.
1.2. INTERLEAVINGS, SCHEDULES, AND COMPUTATION TREES

Figure 1.8: Computation tree for TwoDiningPhilosophers

between states as tree arcs.

Figure 1.7 displays the initial portion of the computation tree for the SumToN system. The orange dashed line represents the first several steps of the schedule on the left side of Figure 1.6, and the green dotted line similarly represents a prefix of the schedule on right side of Figure 1.6. The initial state of SumToN is the root of the tree; we will henceforth refer to it as $s_0$.

Notice that many different tree arcs emanate from $s_0$. This means that multiple transitions are enabled. A moment’s reflection on the BIR code for SumToN reminds us that all three threads are marked as initially runnable (active) – this gives rise to the fact any of the three could be the first to execute. We call such a point where the schedule must choose between several enabled transitions from different threads a choice point. In this particular example, each of the threads Thread0, Thread1, and Thread2 has an enabled transition at $s_0$, represented by the three arcs leading out of $s_0$. Schedule (a) of Figure 1.6 represents a case where transition $1:0 \rightarrow$ is taken, whereas Schedule (b) of Figure 1.6 represents a case where transition $2:0 \rightarrow$ is taken.

The computation tree of Figure 1.7 clearly illustrates that there may be many schedules for the SumToN system. Of course, in any real implementation of such a system, the actual scheduling strategy would be determined by the implementing language’s underlying run-time system or the underlying operating system. Thus, the particular scheduling strategy would likely vary across different implementations and platforms. Moreover, in distributed systems (which we will see later in subsequent chapters), there is even less control of the ordering of instruction execution across processes.

In formal reasoning about concurrent programs, one often abstracts away from implementation or platform details including such issues as the particular scheduling policy used. This allows one to make conclusions about the correctness of a modeled system when it is deployed in any context. In the following sections, we will see the mechanisms that a tool like Bogor provides for reasoning about some or all of a system’s possible schedules.

1.3 Executing (Simulating) BIR Systems with Bogor

The examples and exercises of the previous section clearly illustrate that, even for quite small concurrent systems, it is very difficult to reason about possible instruction interleavings. As a first step towards an exhaustive search of a system’s schedules, many model-checkers provide simulation facilities for analyzing the system being developed. These simulation facilities can be used in two modes – random simulation or guided simulation – that differ in the manner which an instruction is chosen for execution at a choice point. In a random simulation, when the simulator encounters a choice point, it randomly selects the next instruction to execute from the set of enabled transitions. In a guided
simulation, when the simulator encounters a choice point, the simulator typically displays a list of all the enabled transitions at that point, and the user determines the next instruction to execute by picking from the list.

Now we turn to Bogor’s guided simulation mode. Figure 1.9 displays the result of invoking Bogor’s guided simulation mode on the TwoDiningPhilosophers example of Figure 1.1. Referring to the computation tree in Figure 1.8, Figure 1.9 (a) illustrates that the simulator has executed all the transitions (in our example, zero) up to the first choice point $s_a$ and has paused and is waiting for the user to select from the two enabled transitions at that point. Figure 1.9 (b) shows the Bogor display that results if transition from Philosopher1 is chosen (i.e., $P_1:0 \rightarrow$ is taken). The simulation continues along the schedule determined by the user in this way until a state is reached where there are no more enabled transitions or until an assertion is violated.

The discussion above and the exercises below reveal both the benefits and the frustrations that can result from using a model-checker’s simulation capabilities. In random simulations, the model-checker may sometimes find an error but often it will fail to pick a schedule that leads to a defect in the model. Guided simulation is useful because it automates a process that developers often carry out in their minds or on paper – namely, stepping through a program in a particularly tricky region in an effort to gain more insight into system’s execution (e.g., possible interweavings and values of variables). Using guided simulation, the user may guide the model-checker to a property violation, but generally this requires that the user already have a good idea about how the property violation can occur. Guided simulation is tedious but can be effective for short traces. It is less effective or even infeasible for longer traces. It certainly cannot be used in practice to obtain an exhaustive search of all possible schedules.

**Exercises**

1. **(Guided)** Set the parameter $N$ of the SumToN system to various values and analyze the system using Bogor’s random simulation mode. Is Bogor able to find any violating traces in random simulation mode?

2. **(Guided)** Set the parameter $N$ of the SumToN system to 5, and construct a violating trace using Bogor’s guided simulation mode. Record the schedule path using the transition notation introduced earlier in the text. Is this the shortest trace that leads to a violation of the assertion? How can you be sure?

3. Set the parameter $N$ of the SumToN system to 7, and construct a violating trace using Bogor’s guided simulation mode. Record the schedule path using the transition notation introduced earlier in the text. Is this the shortest trace that leads to a violation of the assertion? How can you be sure?
1.4 Verifying BIR Systems with Bogor

In the previous section, we saw that random simulation may be useful for “tire-kicking” – a quick and superficial check to assess the quality of a model, but since it only explores one execution trace it’s likely not to be that useful for finding bugs. We also saw that guided simulation is useful when trying to determine the possible execution traces for a particularly tricky section of the system, but since much manual effort is required, it’s not very effective for exploring a lot of traces or traces that are quite long.

The main strength of state exploration tools like Bogor is not their simulation capabilities, rather it is their ability to perform exhaustive searches of a system’s state space. In this section, we introduce the core algorithm used in explicit state model checkers. We will orient the presentation around the primary data structures used by the algorithm. In fact, there are a variety of search strategies that have been implemented in explicit state model checkers. By far, the most common is depth-first search (DFS). We will focus on the basic depth-first search algorithm in this section; other types of searches will be discussed in later chapters.

Recall that the “interesting aspects” of the simulation algorithms of the preceding section related to how the algorithms proceeded when they arrived at a choice point where more than one transition was enabled: random simulation picked one of the enabled transitions at random, whereas user-guided simulation asked the user to select a transition. In an exhaustive depth-first search, the system’s entire computation tree is explored. When a choice point is encountered, the model-checker selects an unexplored transition, but it also keeps track of the fact that it needs to return to that point and explore the remaining transitions leading out of that point.

The basic depth-first search algorithm

Figure 1.10 gives an overview of a depth-first search of the SumToN system. Figure 1.10 (a) illustrates that, as the algorithm completes the exploration of the entire sub-tree leading out of the left-most transition of s, it back-tracks until it finds a state where transitions remained to be explored (s, in this case), and then it continues with next unexplored transition leading out of s. Once all paths out of s have been explored, the algorithm back-tracks to s and continues exploring the unexplored paths from that point. Figure 1.10 (b). The process continues as illustrated in Figure 1.10 (c) until the entire tree is covered (not pictured).

Figure 1.11 presents the outline of the basic DFS algorithm. This algorithm uses three different data structures:

- a state vector s holds the value of all variables as well as program counters for each thread,
- a stack of state vectors holds the states encountered down the current path in the computation tree, and
Figure 1.2 of the previous section illustrates the sequence of state vectors for
will terminate when a seen set is used. SumToN assures us that the DFS on
vector, stack and seen set structures to obtain a highly optimized algorithm with respect to space and speed. We will ... the values of the data structures in a simple abstract manner that captures the essences of the issues we wish to present.

overflows. This
x
be a back edge in the computation graph when the value of
Model-checkers usually employ multiple clever representations of the state
type). In this case, there will eventually
contains a finite number of states since it has (1) only statically allocated threads, each of which has a fixed number of possible control points and (2) global variables each fixed at 8 bits long (the byte type). In this case, there will eventually be a back edge in the computation graph when the value of x overflows. This assures us that the DFS on SumToN will terminate when a seen set is used.

Figure 1.11: Depth-first search state-space exploration

1  seen := \{
2  stack := cons(s0, nil)
3
4  while stack ≠ nil do {
5      s := head(stack)
6      stack := tail(stack)
7      if s /∈ seen {
8          checkState(s)
9          seen := seen ∪ \{s\}
10          workSet := envSel(s)
11          for all α in workSet do {
12              s′ := α(s)
13              stack := cons(s′, stack)
14          }
15      }
16  }

• a seen state set holds the state vectors for all the states that have been checked already (i.e., seen) in the depth first search.

The algorithm of Figure 1.11 proceeds as follows. First, seen is initialized as the empty set, indicating that no states have yet been visited. Next, the initial state is placed on the stack of states which must still be visited. The stack will serve an important purpose: when the end of a path in the computation tree is reached, the topmost stored state on the stack indicates the state to which the exploration should backtrack and restart.

Each iteration of the while loop at line 4 of Figure 1.11 starts by fetching the next state s to be analyzed. If the subtree starting at s has already been seen, the iteration aborts immediately. Next, the state is checked to see if all invariants and assertions are satisfied by invoking checkState(s). By inserting the current state into the seen set at line 9, we ensure that the algorithm will never explore a given state more than once. Next, all the outgoing transitions from s that are enabled are collected in workSet. Then, for each transition α in the workSet, α is applied to the current state s to get a next state s′ along the current path. The new state s′ is then pushed onto the top of the pending-work stack; a future iteration of line 4’s while loop will explore the entire subtree rooted at s′. As a consequence, all paths leading out of s are explored.

Model-checkers usually employ multiple clever representations of the state vector, stack and seen set structures to obtain a highly optimized algorithm with respect to space and speed. We will discuss optimizations in later chapters, but for now we represent the values of the data structures in a simple abstract manner that captures the essences of the issues we wish to present.

Figure 1.2 of the previous section illustrates the sequence of state vectors for a particular trace of the example in Figure 1.5.

We now provide some additional intuition about the role of the seen set. Often it is the case that the DFS algorithm will proceed down a path in a system’s computation tree and it will encounter a state s that has already been encountered before on a different path through the computation tree. In such a case, there is no need to check s again (or any of s’s descendant states in the computation tree) since these states have already been checked previously. In the DFS algorithm, the seen set holds the states that have been seen before, and the algorithm 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.

For example, consider the fragment of the computation tree presented in Figure 1.10. Assume that the DFS algorithm has already explored the left path of the computation tree and is now proceeding down the right path. When it encounters the state s0, it finds this state in the seen set. This causes the algorithm to backtrack to begin exploring other unexplored paths.

In cases where a transition along one path leads to a state s that has been encountered on another path, it is sometimes convenient from a conceptual standpoint to view s as being shared between the two paths as illustrated in Figure 1.12. In this view, the computation tree is actually represented as a computation graph, where a node that has more than one incoming arc represents a state that is encountered multiple times during the DFS search.

Due to the use of the seen set, checking a system with non-terminating executions (i.e., the system’s computation tree contains infinite paths) may actually terminate if the system has a finite number of states. Our SumToN example contains a finite number of states since it has (1) only statically allocated threads, each of which has a fixed number of possible control points and (2) global variables each fixed at 8 bits long (the byte type). In this case, there will eventually be a back edge in the computation graph when the value of x overflows. This assures us that the DFS on SumToN will terminate when a seen set is used.
1.4. VERIFYING BIR SYSTEMS WITH BOGOR

Exercises

The goal of these exercises is to help the reader understand the depth-first search state-space exploration algorithm, and in particular, the data structures associated with the algorithm.

1. (GUIDED) Given the computation tree in Figure 1.8 for the dining philosopher system of Figure 1.1, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the yellow-encircled labels in the computation tree.

2. (GUIDED) For the bounded buffer example of Figure 1.3, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).

3. For the bounded buffer example of Figure 1.3, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).

4. For the readers/writers example of Figure 1.4, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).
3. Bogor Architecture

3.1 SSEARCH: State Space Exploration

4. Extensions

4.1 Language Extensions

4.1.1 Syntax Extensions

4.1.2 Semantics Implementation

4.2 Module Extensions

A. BIR Reference

A.1 Syntax

A.2 Semantics

A.3 Keywords

B. Bogor API

C. API Reference

D. Module Reference

E. Model Checking BIR Models

F. Counter-example Display

G. Interactive Mode

H. User-Guided Mode

I. Command-Line Interface

Table of Contents

1. Introduction

1.1 Bogor Overview

1.2 Installing Bogor

1.3 Using Bogor

1.3.1 Model Checking BIR Models

1.3.2 Counter-example Display

1.3.3 Random Simulation Mode

1.3.4 User-guided Simulation Mode

1.3.5 Configuring Bogor

1.3.6 Bogor Command-line Interface

2. BIR: Syntax

2.1 Systems

2.2 Identifiers

2.3 Types

2.4 Literals

2.5 Constants

2.6 Enumerations

2.7 Records

2.8 Extensions

2.9 Global Variables

2.10 Threads and Functions

2.11 Virtual Tables

2.12 Functional Expressions

2.13 Expressions

Copyright © 2004 SAnToS Laboratory, Kansas State University
Chapter 4. Extensions

Chapter 4. Extensions

Table of Contents
Language Extension  Syntax Extension
Semantics Implementation
Module Extension

Language Extension

In this section, we will use the design, specification, and implementation of `set` native to the BIR language as a running example of enhancing Bogor's input language. This simple, yet illustrative, example highlights both the reasons why one would wish to add direct support for an abstract datatype to BIR and the Bogor infrastructure used to achieve this effect at the cost of little developer time.

Why, one may ask, is the ability to implement complex datatypes as native BIR constructs so important? Occasionally, the model writer wishes to peek "under the hood" of the model checking engine and inspect some property of the system state. In the best case, this is extremely unkindly to do directly in BIR (and usually requires instrumenting the model with auxiliary variables, thereby inflating the state space). Often, the property at hand is simply inexpressible in BIR. For example, one may wish to query if a heap-allocated object is reachable from references beginning at some distinguished object. There is no reasonable way to do this directly in the BIR language; yet, an extremely simple language extension operation implemented with perhaps 2030 lines of Java code can easily decide this reachability query.

More importantly, language extension types and operations enable a tremendous amount of abstraction. To take our running example (a mathematical set), the elements must be kept inside of some container type. Absent the language extension mechanism, this could be directly done using BIR arrays. But then the modeler must confront the usual problems associated with implementing set container types: ensuring element uniqueness and expanding the underlying storage (here, the BIR array) as the set grows are but the most obvious ones. Last, demands unique to model checking impose themselves. Mathematical sets are unordered, but arrays in model checking input languages are not. If the ordering of two array elements is reversed, a model checker will consider these states to be distinct, and in both cases the entire state space descending from these semantically equal, but syntactically different, set encodings will be traversed. This forces the model writer to use some technique for imposing a canonical ordering on the elements. All this is possible in the BIR language, but at the prices of steep runtime penalty (both in memory requirements for the now-larger seen state set, and in time to interpret the added command sequences which manipulate the set data structures) and great confusion of the code expressing the actual domain problem to be checked. By introducing new, encapsulated BIR native datatypes and operations to manipulate them, this code and complexity can be pushed into the model checker runtime environment.

Syntax Extension

In our example, we will define a set abstract datatype which supports the following operations:

- insertion of elements,
- removal of elements,
- emptiness test,
- automated evaluation of predicates across all elements, and
- nondeterministic selection of a contained element.

Informing the Bogor language recognizer about the new datatypes is quite easy. In contrast to the input mechanism of some model checkers (e.g., SPIN) that require modification of the parser itself if new native types must be added, the BIR language allows use of a simple extension declaration block which specifies both the identifiers used for the new types/operations and strong typing information about the arguments and return values of said operations. We show the declaration of new primitives for the set abstract datatype in Figure 4.1, "Declaration of set ADT and operations".

Figure 4.1. Declaration of set ADT and operations

```java
extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule {
    typedef type ('a);
    expdef Set.type ('a) create ('a ...);
    expdef 'a choose ('a) [Set.type ('a)];
    expdef boolean isEmpty ('a) (Set.type ('a));
    actiondef add ('a) (Set.type ('a), 'a);
    actiondef remove ('a) (Set.type ('a), 'a);
    expdef boolean forAll ('a) ('a -> boolean, Set.type ('a));
}
```

The first line declares the extension namespace, Set. We do so by using the keyword `extension` followed by the name of the extension, Set. Then, we use the keyword `for` to specify the Java class which provides the semantics implementation. We will return to a discussion of their implementation in the section called "Semantics Implementation".

Next comes the declaration of the new native type itself. This identifier, `Set.type`, is the type of any set ADT variable which will be declared later. Note that the type `Set.type` (and all operations to follow) are parameterized by `type variable` (here, `'-a'`) set between angle brackets. This allows us to write our set implementation in a completely generic manner so that it is not hard-wired to contain only a specific type of elements, e.g., `int`. Users familiar with the C++ template mechanism or SML's polymorphic type system will quickly recognize the syntax for polymorphism in BIR.

When a Bogor abstract type is declared using placeholder type variables (`'-a'`), it can be instantiated in
a model with any other legal type used as elements. For instance, the following BIR fragment declares

    Set type<int> anIntSet;
    Set type<string> aStringSet;

If we were not interested in making the Set type and operations generalized, we could instead just
declare the type as follows:

    extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule
    {
        typedef type;
        expdef Set.type create(int ...);
        expdef int choose(Set.type);
        ...
    }

This would in turn force us to hard-wire the parameter types of the operators create, add, remove, etc. (we have shown this done with int). As we will see later, supporting polymorphism of extensions
requires some additional code in the Java modules which implement the set semantics. The generality
and robustness gained, though, are usually worth the extra time spent up-front.

The remaining lines in Figure 4.1, “Declaration of set ADT and operations”, are used to declare the
operations allowed on Set.type ADT. These come in two types, split along the same lines as regular BIR
constructs. Expressions are mandated to be side-effect-free; operators which serve as expressions are
declared using the expdef keyword. Actions transform the state of a BIR system; we declare the
operations which perform state changed using the actiondef keyword.

Two operations in particular deserve special mention. First, the expression choose

    nondeterministically retrieves a element from the set. We will see later how Bogor's scheduler can be
made to exhaustively retrieve each element from the set so that the state space traversal explores every
possible outcome from a choose expression. Second, the forAll expression demonstrates Bogor's first-class support for functions. The first argument (whose signature is 'a' => boolean) is the name of a function which accepts the set's contained element type and returns a yes/no answer. We will see
that the implementation of forAll applies the function named as this argument is applied to each
element of the set as a predicate; the operation returns true if and only if the predicate holds on every

At this point, we have shown all the syntax necessary to declare extension datatypes in Bogor. The
markup from Figure 4.1, “Declaration of set ADT and operations” is enough to enable Bogor's type-

checkinig mechanism to verify that the extensions have been used safely. We have, though, only given
an informal description of the intended behavior of the set operations. In the next section, we will
describe the process of writing Java implementations of these operations to fix their semantics.

Semantic Implementation

Once the syntax for the extension has been defined and included in the model, we now shift our
attention to implementing the module which provides the extension's functionality. Recall that, as
shown in Figure 4.1, “Declaration of set ADT and operations”, we name the Java class which
implements a Java language extension's operations:

    extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule
    {
        ... 
    }

We are responsible, then, for providing a Java class SetModule which exposes one method for each
operation (both the actiondef and expdef varieties) named in the extension. The Bogor framework
includes a Java interface, IModule, which any class providing a language extension module must
implement. We thus begin by creating a class SetModule which is tagged as conforming to the
IModule interface:

    package edu.ksu.cis.projects.bogor.module.set;
    public class SetModule implements IModule
    {
        ... 
    }

Because we have tagged our Bogor module class as implementing IModule, we must provide an
implementation for each of its required methods (shown in Figure 4.2, “Required methods for a Bogor
extension module (IModule.java)”). Of the three methods shown, only connect usually has a non-
empty body. Intuitively, this is because connect establishes the module's links to the main Bogor
model checking components, while getCopyrightNotice and setOptions are sometimes used to display legal messages and configure advanced options.

Figure 4.2. Required methods for a Bogor extension module (IModule.java)

    public interface IModule extends Disposable
    {
        String getCopyrightNotice();
        void setOptions(String key, Properties options);
        void connect(IBogorConfiguration bc);
    }

We will follow suit, making getCopyrightNotice return the null string and setOptions do
nothing. We will define a connect method that acquires references to some of the Bogor runtime
components, and a dispose method required by the fact our IModule implementation is transitively
required to implement Bogor's Disposable interface too:
// acquire references to runtime modules
public void connect(IBogorConfiguration bc) {
    tf = bc.getSymbolTable().getTypeFactory();
    ee = bc.getExpEvaluator();
    ss = bc.getSchedulingStrategist();
    vf = bc.getValueFactory();
}

// release references to runtime modules
public void dispose() {
    tf = null;
    ee = null;
    ss = null;
    vf = null;
}

In our case, we have chosen to store references to Bogor's type factory, main expression interpreter, scheduling policy, and value creation mechanism. This selection is dictated by the subset of functionality we'll need to maintain the set representation. In principle, an extension module can connect to any or all of the nine core modules. If in doubt, we suggest that extension writers initially not save any references to Bogor modules. These can always be added later during development as needed.

Operations

The code shown so far is a minimal (i.e., zero functionality) Bogor extension. It can be loaded into the runtime and acquire model checker component references, but because no additional methods beyond those required by IModule have been defined, it cannot manipulate any extension types. We must add one Java method into the SetModule class for each operation (e.g., create, choose, and add) shown in Figure 4.1, "Declaration of set ADT and operations".

Recall that every extension operation is either an expdef or an actiondef. The Java method implementing each of these two varieties of operations takes a fixed signature. In Figure 4.3, "Method signatures for extension operator implementations", we see the pattern for writing both action and expression operator implementations.

Figure 4.3. Method signatures for extension operator implementations

    // expdef variety
    public IValue exp0OperationName(IExtArguments args)
    {
    ...
    }

    // actiondef variety
    public IBacktrackingInfo[] actionOperationName(IExtArguments args)
    {
    ...
    }

This provides the signature of the method. But how is the value which is stored in IBRSet.type'<a>variables produced? We will discuss strategies for writing Java code to represent this value type later in the section called "Value Types" and for now just note that there are significant differences between the methods one uses to handle IValue objects representing Bogor primitives types (e.g., numerical data and enumerated types) and Bogor reference types (e.g., strings, records). The bulk of create's body is devoted to detecting which type of element the set will contain and adjusting appropriately. We show here the entire method body except code used to detect and adapt to primitive element types (it has been elided as noted with an ellipsis):

    Type argType = (Type) arg.getTypeVariableArgument(0);
    Type setType = arg.getExpType();
    if (argType instanceof NonPrimitiveType)
    {
      // builds an empty set
      ISetValue result = new ReferenceElementSetValue((NonPrimitiveExtType)
        arg.newReferenceId());
      // add the arguments to the set
      int size = arg.getArgumentCount();
      ...
for (int i = 0; i < size; i++)
{
    result.add(arg.getArgument(i));
}
return result;
} 
else if (argType instanceof PrimitiveType)
{
    ... 
} 
else 
{
    assert false;
    throw new RuntimeException("Unimplemented extension");
}

We inspect the type of elements which will populate the set by retrieving the first type variable (our operation uses only one: ‘<‘). The inheritance hierarchy for Bogor types (rooted at edu.ksu.cis.projects.bogor.type.Type) shows that every value-bearing types descend from either PrimitiveType or NonPrimitiveType. Because all non-primitive types are allocated on the heap, and thus variables of these types are references, we group them together and use a particular set IValue implementation suited to manipulating contained elements as reference types. In the code we have elided, a different set implementation able to cope with primitive-typed elements is constructed and returned instead.

**Note**

Pleasantly, we are able to localize the tedious inspection of element types to only the create method. The block of code dealing with PrimitiveType is able to configure its dedicated ISetValue implementation to autonomously deal with all the different Bogor primitive types. So, the rest of our operation implementations are much tidier to list.

After the reference-type-specific set value is created, we just iterate over all the arguments (initial elements of the set) and add them one by one to the set implementation. The ISetValue.add (IValue) method is sufficient to do this. Finally the set value is returned as an IValue to complete the operation.

Next, consider the choose expression. This leverages the functionality of Bogor's scheduling module to non-deterministically pick and return an object from the set. Recall the listing of its type and arguments from Figure 4.1, “Declaration of set ADT and operations”:

```java
expdef 'a choose<'a>(Set.<'a>a>>;
```

As an expression operator, then, choose will be implemented by a Java method in ISetModule.java with the following signature:

```java
public IValue choose(IExtArguments arg)
{
    ... 
}
```

The procedure for nondeterministically picking an element from set turns out to be surprisingly simple. Here is the body of choose:

```java
// gets the elements of the set
ISetValue set = (ISetValue) arg.getArgument(0);
IValue[] elements = set.elements();
int size = elements.length;
int index = 0;
if (size > 1) {
    // ask the scheduler which one should be picked now
    index = ss.advise();
    arg.getNode(),
    arg.getScheduleInfo());
}
// returns the one picked by the scheduler
return elements[index];
```

1. The set.elements method returns the contained objects in some canonical order, so that the *i*th element of the array is the same each time that the transition is executed.
2. If the set only contains one element, nondeterminism is not needed.
3. This seems a little like “magic.” When the advise call is made on the scheduler ss, Bogor interprets this as a request to branch the state space exploration. Thus advise will successively return the values 0, 1, ... elements.length - 1.

The first time advise is executed, the scheduler returns 0 and makes a note that once the DFS bottoms out and backtracks, the current transition (from which choose was evaluated) should be replayed. The next time that advise is consulted, it returns 1. Again the ensuing DFS again bottoms out and backtracks to the current transition. This continues, until advise returns elements.length - 1. At this point, the scheduler notes that all possible members of the elements array have been explored. When the backtracking following the elements.length - 1 response moves back past the transition using choose, the scheduler simply declines to order an additional DFS. The non-deterministic exploration is complete.

The implementation of the next operation, isEmpty, is extremely easy. The novel technique to observe is the use of the value factory (IValueFactory) to create a BIR value wrapping the boolean result. After extracting the set from the IExtArguments passed down from Bogor's runtime, the helper method getBooleanValue uses the value factory to create a new boolean IValue:

```java
public IValue isEmpty(IExtArguments arg)
```

---

Chapter 4. Extensions

Next is the most complicated expression we'll see: forAll, which applies a predicate to all elements of the set. The signature for the BIR expression fragment `forAll:ADT defects operations` was this:

```java
expdef boolean forAll='<a>('a -> boolean, Set.type<a>);
```

While the description of `forAll`'s semantics sounds intimidating, we are able to delegate most of the work to Bogor's expression evaluator module to determine if the predicate holds on each element. Our job then, is just to iterate across all the elements contained in the set:

```java
public IValue forAll(IExtArguments arg)
{
    // get the fun name of the function
    String funName = ((IStringValue) arg.getArgument(0)).getString();

    // get the set elements
    ISetValue set = (ISetValue) arg.getArgument(1);
    IValue[] elements = set.elements();

    // assume all true
    boolean result = true;

    // for each element, apply the function
    // if it returns false for at least one element, then there exists
    // an element that does not satisfy the condition
    for (int i = 0; i < elements.length; i++)
    {
        IValue element = elements[i];
        IIntValue val = (IIntValue) element.evaluateApply(
            funName,
            new IValue[] { element });
        if (val.getInteger() != 1)
        {
            result = false;
            break;
        }
    }
    return getBooleanValue(result);
}
```

Chapter 4. Extensions

What's going on here? The function's type is listed as`<a -> boolean, List<String>`. Because functions are not simply values which can be copied around, the predicate is transmitted to the Java implementation of `forAll` by sending the name of the function's identifier. Once "under the hood," we can use the expression evaluator module to evaluate the function just by passing its name and the argument(s).

Bogor, like the Java Virtual Machine, encodes boolean values as 0- or 1-valued integers. So the result of the boolean predicate is communicated back as an `IIntValue`.

Another use of the value factory to create a wrapped primitive value suitable for Bogor's interpreter engine.

This completes the complement of expression operators we have specified on the set extension. Next, we take a look at the techniques for implementing destructive-update operations (in BIR terms, `actionNode(s)`). The Java code providing a BIR action is fundamentally different than its expression cousins: rather than returning a value, the method must supply "undo" objects which can be executed to return the BIR system to the state just before the action was taken.

Our set extension to the BIR language only includes two side-effecting operations: `add` and `remove`. Conceptually, both are extremely simple and very nearly mirror operations, so we will only examine the former, `add`.

Our basic strategy for implementing the set insertion operator is to case-split. If the "new" element is already in the set, then execution aborts immediately and no undo information is needed (we have made no change to the state). If, on the other hand, the proposed addition is valid, then we insert the element into the set and return an `IBacktrackingInfo` object which, when executed, will remove the new element from the set. The code is given below.

```java
public IBacktrackingInfo[] add(IExtArguments arg)
{
    // get the set
    ISetValue set = (ISetValue) arg.getArgument(0);

    // get the element to be added
    IValue element = (IValue) arg.getArgument(1);

    if (!set.contains(element))
    {
        // add the element
        set.add(element);
        ISchedulingStrategyContext ssc = arg.getSchedulingStrategyContext();
        // create the backtracking infos
        return new IBacktrackingInfo[] {
            createAddBacktrackingInfo(set, element),
        };
    }
    return null;
}
```
The exact procedure for testing whether an element already belong to the set depends on the static type of the element (e.g., int versus a reference type). Each ISetValue implementation will define isContains method in a way meaningful to its domain.

The real work of creating the undo operation object is delegated to a helper function.

While it demonstrates how to add elements to a set, this code listing doesn't really help us to understand the key problem: how to create the IBacktrackingInfo undo operations. To see this, we have to look behind the implementation of the helper function. Here, then, is the complete body of createAddBacktrackingInfo in full glory:

```java
protected IBacktrackingInfo createAddBacktrackingInfo(
    final ISetValue set,
    final IValue element,
    final Node node,
    final int stateId,
    final int threadId,
    final ISchedulingStrategyInfo ssi)
{ return new IBacktrackingInfo()
    | public Node getNode()
        | return node;
    | public ISchedulingStrategyInfo
        getSchedulingStrategyInfo()
        | return ssi;
    | public int getStateId()
        | return stateId;
    | public int getThreadId()
        | return threadId;
    | public void backtrack(IState state)
        | set.remove(element);
    | }
}

public IBacktrackingInfo clone(Map cloneMap)
{ return createAddBacktrackingInfo( (ISetValue) cloneMap.get(set),
    (IValue) (element instanceof
INonPrimitiveValue)
    ? cloneMap.get(element)
    : element),
    node,
    stateId,
    threadId,
    ssi.clone(cloneMap));
}

public void dispose()
{ }
}
}
```

Most of the IBacktrackingInfo methods just pass along simple data such as (1) the syntax tree node for the actionset executed (2) a process descriptor recording which BIR thread executed the code (3) and the state vector ID before the action executed. Only two methods need special attention. First, backtrack is the actual "undo" undo operation. In this case, it just needs to remove the element inserted by add. Second, the close method; this one's purpose is a little more indirect. Although this method is not used in the current Bogor codebase, it would be used if a future version of Bogor cloned the system state (IState); because the object references in a deep state clone would change, a clone of the backtracking operation would be necessary also in order to operate on this new forest of references.

Value Types

We have, so far, engaged in vigorous hand-waving about the nature of the value types in BIR language extensions. Most notably, while describing the implementation of the create expression operation, we hinted at the fact that any one of several different classes may implement the same abstract Set.type<‘a> type. In this section we will examine Bogor's mechanism for grafting new nodes onto the internal type hierarchy. We shall also see how to a bridge between (1) low-level storage of values and (2) high-level model checking requirements such as state vector creation, heap ordering, and garbage collection.

Figure 4.4. Required methods for any Bogor ADT extension type (INonPrimitiveExtValue.java)

```java
public interface INonPrimitiveExtValue
    extends INonPrimitiveValue, Serializable
{
    // Methods required directly
    void externalize( PrintWriter pw,
        INonPrimitiveValueIdTracker npIdTracker);
```
byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMap,
    int bitsPerThreadId,
    IntIntTable threadOrderMap);

void visit(
    IValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IValueVisitorAction vva);

// Methods required by INonPrimitiveValue
int getReferenceId();
// Methods required by IValue
Type getType();
int getTypeId();
IValue clone(Map cloneMap);
void validate(BogorConfiguration bc);
}

Figure 4.5. Required operations for all set types (ISetValue.java)

public interface ISetValue extends INonPrimitiveExtValue {
    void add(IValue v);
    boolean contains(IValue v);
    IValue[] elements();
    boolean isEmpty();
    void remove(IValue v);
}

All value classes in Bogor must implement the IValue interface. Usually, this is done indirectly by implementing one of its descendants. There are two value interfaces in particular which extension types may choose to implement: IPrimitiveExtValue and INonPrimitiveExtValue. In our case, a set is certainly not a primitive atomic type. Rather, it is an abstract datatype that contains other objects. As such, our Bogor value class must implement INonPrimitiveExtValue (the required methods are shown in Figure 4.4 “Required methods for any Bogor ADT extension type (INonPrimitiveExtValue.java”). The interface requires three methods: linearize, visit, and externalize, as well as those that its parent interfaces require. We will study the example implementations of these methods in ReferenceElementSetValue.java.

As an exercise in good design methodology, we have also created a new Java interface, ISetValue, which specializes the INonPrimitiveExtValue type. See Figure 4.5, “Required operations for all set types (ISetValue.java)”, for a listing of its required operations. Our running example, ReferenceElementSetValue.java is an implementation of ISetValue.

The first of these, linearize, is the critical bridge from the Java world of objects, scalar types, and hierarchical structures to the model checker’s state vector. Recall that model checkers use a seen-before set to record which states in a computation tree have already been visited. This is vital to the depth-first space space exploration, because it provides a means to prove termination when the state space is finite. Algorithms listings usually abstract away from the implementation of this set. In Bogor, the seen-before set is a collection of bit sequences which encode states. A state is said to be in the seen-before set if and only if the seen-before set contains a bit sequence equal to the bit sequence which encodes s. The linearize method is tasked with walking the internal structure of an extension type and producing a sequence of bits which uniquely encodes the state information about the datatype.

Because a typical state space exploration will accumulate tens of thousands or more distinct states’ bit-vectors in memory simultaneously, it is imperative that the bitwise encoding produced by linearize is efficient. Let’s glance at the signature:

```
byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMap,
    int bitsPerThreadId,
    IntIntTable threadOrderMap);
```

The return value is a sequence of byte arrays (normally, the outer dimension in only one deep). The first and third arguments give the minimum number of bits needed to encode process descriptors and heap location identifiers. The second argument gives a mapping from the value objects representing heap-allocated types a unique identifier for each such object. We will use this shortly. In the ReferenceElementSetValue version of this method (its body is shown next), note how the unique object ID is fetched for each set element. After sorting these IDs (because sets are unordered and we wish our linearization to be a symmetric set), the bit-vector is constructed by just concatenating the binary representation of all these object IDs.

```
public byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMap,
    int bitsPerThreadId,
    IntIntTable threadOrderMap) {
    IValue[] elements = elements();
    BitBuffer bb = new BitBuffer();
    int[] elementIds = new int[elements.length];
    for (int i = 0; i < elements.length; i++) {
        elementIds[i] = nonPrimitiveValueIdMap.get(elements[i]);
    }
```
Arrays.sort(elementIds);
for (int i = 0; i < elements.length; i++)
{
    bb.append(elementIds[i],
        bitsPerNonPrimitiveValue);
}
return new byte[][] { bb.toByteArray() };

1. The unique ID of each set element object is retrieved. It is this value which gets written to the state vector.
2. Note the use of the ByteBuffer to handle construction of bitwise encodings. Using its convenience methods is the usual idiom when writing a linearization method.
3. We must specify the number of bits to use when encoding the element ID into the bit vector. It happens that Bogor has pre-calculated this for our particular element type, but usually this must be manually calculated withUtil.widthInBits().

The contents of the two-dimensional byte array which we return are appended to the overall bitvector, and a linearized representation of the overall system state is created. One factor important to correctly implement a datatype linearization is to always impose an outside ordering on the elements if their order of occurrence in the container is not relevant. If we implemented a queue extension type, the ordering of elements is important to the semantics; is it after all a first-in, first-out data structure? But the semantics of a set care not whether the most recently inserted element is stored at the end of the containing container or at the beginning. To prevent these two conditions from artificially producing different bit vectors, we first impose a total ordering on the elements by sorting according to their object ID's.

Next we move to the visit method. This method is called as part of an overall traversal that collects all IValues in the Bogor system. This is needed for several analytical purposes: garbage collection, obtaining canonical representations of the heap, etc. Its signature is given by the following listing:

```
public void visit(
    final IVValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IVValueVisitorAction vva)
{
    ... 
}
```

The interesting bit about visit is that the traversal which it helps to implement is ordered. First, the search can be either a pre- or post-order traversal, as controlled by the depthFirst flag. A pre-order traversal will set this flag to true, this means that all IValues contained inside the our ADT should be prepended to the beginning of the workList queue. If, on the other hand, the flag is false, this indicates a post-order traversal; all contained values should be appended to the end of the workList queue.

Second, the elements themselves should be inserted in a particular order. If our set contained BIR primitive values as elements, then we could use the ValueComparator argument as a means to sort them. Because the set implementation we consider here stores references values, though, we must sort them in an ad hoc manner. For our purposes, the ordering by heap ID already done by the elements() method call is good enough to ensure a symmetric representation. This done, we simply add the elements to the work queue according to whether the traversal is depth-first or breadth-first:

```
public void visit(
    final IVValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IVValueVisitorAction vva)
{
    IVValue[] elements = elements();
    if (depthFirst)
    {
        for (int i = 0; i < elements.length; i++)
        {
            workList.addFirst(elements[i]);
        }
    }
    else
    {
        for (int i = 0; i < elements.length; i++)
        {
            workList.add(elements[i]);
        }
    }
}
```

The last of the methods required for all nonprimitive extension types (as shown in Figure 4.4, "Required methods for any Bogor ADT extension type (NonPrimitiveExtValue java) is externalized. Here, the abstract contents of the data structure are written to an output stream (usually in the form of XML). The implementation of this method is not interesting; the reader can consult ReferenceElementSetValue.java for a representative example.

Finally, we still must implement the methods required by the set interface itself (shown in Figure 4.5, "Required operations for all set types (SetValue java)." Our set implementation uses a standard Java Collections java.util.HashSet to store its elements, because by happenstance there is a one-to-one correspondence between BIR heap-allocated objects and the Java object instances which implement them. This means that the default java.lang.Object.equals() method that compares Java object references is sufficient to preserve uniqueness of BIR objects, and we can use a Collections set as a convenient container. Thus we add a set container as a field to our ReferenceElementSetValue class:

```
protected HashSet set = new HashSet();
```
Of the five required `SetValue` methods, four are now reduced to one-liners: `add`, `contains`, `isEmpty`, and `remove`. They just delegate the work to the `HashSet` container. Because all four are substantially similar, we just show the implementation of `add`:

```java
public void add(IValue v)
{    set.add(v);}
```

Only the `elements` method has an interesting implementation. Here, we must retrieve the BIR set's elements from the delegated container and sort them by increasing order of their heap identifier:

```java
public IValue[] elements()
{
    Object[] elements = set.toArray();
    orderValues(elements);
    IValue[] result = new IValue[elements.length];
    System.arraycopy(elements, 0, result, 0, elements.length);
    return result;
}
```

```java
public void orderValues(Object[] values)
{
    Arrays.sort(values, new Comparator()
    {
        public int compare(Object o1, Object o2)
        {
            int refId1 = ((INonPrimitiveValue) o1).getReferenceId();
            int refId2 = ((INonPrimitiveValue) o2).getReferenceId();
            return Util.compare(refId1, refId2);
        }
        public boolean equals(Object obj)
        {
            return this == obj;
        }
    });
```

This completes our walkthrough of the procedure and concerns involved when adding a new native datatype to BIR. The complete sources (`SetModule.java`, `ISetValue.java`, `ReferenceElementSetValue.java`, and `PrimitiveElementSetValue.java`) are available in the public GudangBogor repository reachable from `http://gudangbogor.projects.cis.ksu.edu` in addition to being available bundled here:

- `SetModule.java`
- `ISetValue.java`
- `ReferenceElementSetValue.java`
- `PrimitiveElementSetValue.java`
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 00: Overview of Lectures

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS’04

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

Research Context

SAnToS Laboratory, Kansas State University
http://www.cis.ksu.edu/santos

- Research on Static Analysis and Transformation of Software
  - model-checking, static analysis, model-driven component-based development, slicing, partial evaluation, etc.
- Aiming for robust tools
  - open source, close to commercial quality
- Integration into development process
  - ease of use and scalability sometimes take precedence over theoretical elegance
- most of the time, focus is on bug-finding rather than true verification
- Trying to build on lessons learned…
  - …from previous versions of tools (e.g., old Bandera)
  - …from interaction with industrial partners

Research Context -- Bogor

In ESSCaSS’04 lectures, I’ll focus on...

Bogor Model Checking Framework
http://bogor.projects.cis.ksu.edu

• Supporting model-checking of OO software (Java, in particular)
• Open platform for research/experimentation
  - take your favorite new idea, implement it in Bogor to try it out
• Teaching tool
  - foundation of a tool/application-oriented course on model-checking
  - some material already available; much more on the way

Perspective for ESSCaSS’04

• There is a lot of great stuff going on in the area of program verification and software model-checking
• I’ll focus on work that our group has done
  - …you can look in papers referenced as background reading for related work from our group and others
• You’ll get my perspective on software model-checking
  - …there are many other valid perspectives
• Lecture themes
  - Bogor as an extensible model-checking framework
  - Using Bogor to build your own customized model-checker
  - The value of domain-specific model-checking customizations
• Lecture non-themes (very important, but not covered)
  - automata-theoretic view of algorithms, symbolic model-checking (BDDs), details of temporal logic, data/predicate abstraction

Lecture Outline

• Lecture 1: Foundations of Model-Checking
  - Overview of Bogor
  - Overview of basic depth-first search explicit state reachability algorithm
• Lecture 2: Bogor Architecture & Extensions
  - Toward understanding the guts of Bogor and how to modify it
  - How to extend Bogor’s modeling language
• Lecture 3: Representing Java, Checking Specifications
  - Representing Java in Bogor
  - A quick overview of Bogor reduction algorithms
  - Checking JML specifications
  - Checking atomicity specifications
• Lecture 4: Customizing Bogor to Check Avionics Designs
  - Cadena – an IDE for design of component-based distributed systems
  - Checking Cadena designs in Bogor

Tools Used in Lecture

• Bogor installation
  - demos during lectures
  - on your CD, includes user manual, examples, etc.
  - if you use Bogor, we would like you to register on the Bogor web-site!
    - http://bogor.projects.cis.ksu.edu
• Cadena
  - demos during lectures
  - available for download
Supporting Material

- Chapter from our lecture notes of foundations of model-checking
  - in your printed material, on CD
- Tutorial on Writing Bogor Extensions
  - in your printed material, on CD
- Research papers for background reading
  - on CD
- All supporting material on CD can also be found at the following site...
  - http://www.cis.ksu.edu/~hatcliff/ESSCaSS04

Other SAnToS Tools & Material

See your printed hand-out for more information...

- Bandera
  - Java Model-checking Environment
  - next generation due out an end of 2004
- Cadena
  - environment of design, specification, and analysis of distributed component-based systems
- Indus
  - static analysis and program transformation tools (e.g., full Java slicer)
- Course material on software specifications
- Course material on foundations and applications of model-checking
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 01: Bogor Overview

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS’04

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
Lockheed Martin
NASA Langley
Rockwell Collins/RTC
Intel
Sun Microsystems

Bogor - Software Model Checking Framework

Bogor - Direct support for 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 - Eclipse-based Tool Components

Architecture allows encapsulation/integration with other verification tools using IBM’s Eclipse Integrated Development Environment

Bogor - Domain Specific Model-Checking

Modeling language and Algorithms easily customized to different domains

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

Hardware
Device Drivers
Avionics
Telephony
Automotive
GUI

Leveraging Domain Knowledge

- Holzmann developed a customized model extraction from C to Spin
- Translation using pattern matching of particular domain idioms
- In essence, an abstract machine for a particular domain
- Very effective at finding subtle defects

Lucent Path Star Telephone Switch

Variety of System Descriptions

Design Notations
State Machines
Byte code
Source code

Abstract machine tailored to domain and level of abstraction

Model-checking Engine
Domain & Abstraction Extensions

The Goal

Model-checking Engine
Domain & Abstraction Extensions

Abstract machine tailored to domain and level of abstraction
Customization Mechanisms

Bogor -- Extensible Modeling Language

- Threads, Objects, Methods, Exceptions, etc.
- Domain-Specific Abstractions

Core Modeling Language

Bogor -- Customizable Checking Engine Modules

- Scheduling Strategy
- State-space Exploration
- State Representation
- Domain-Specific Search
- Domain-Specific Scheduler
- Domain-Specific State Rep.
- Core Checker Modules
- Customized Checker Modules

Outline

- Bogor Modeling Language and UI
  - Example: Dining philosophers
  - Demo: Bogor UI and BIR Case Wizard
- Concept of Bogor Extensions
  - Extending the syntax
  - Adding semantics via Java
- Conclusions
  - The utility of a customizable model-checking platform

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.

A BIR Example — 2 Dining Philosophers

```
uses a record to model forks
```

```
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;
    }
    goto loc2;
    loc loc1: // take right fork
    when !right.isHeld do {
      right.isHeld := true;
      goto loc3;
    }
    goto loc4;
    loc loc2: // put right fork
    do { right.isHeld := false; }
    goto loc3;
    loc loc3: // put left fork
    do { left.isHeld := false; }
    goto loc0;
    loc loc4: // put left fork
    do { left.isHeld := false; }
    goto loc0;
    loc loc5: // put right fork
    do { right.isHeld := false; }
    goto loc0;
  }
}
```

Overview

Concept of Bogor Extensions

- Extending the syntax
- Adding semantics via Java

An Example — 2 Dining Philosophers

```
uses a record to model forks
```

```
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;
    }
    goto loc2;
    loc loc1: // take right fork
    when !right.isHeld do {
      right.isHeld := true;
      goto loc3;
    }
    goto loc4;
    loc loc2: // put right fork
    do { right.isHeld := false; }
    goto loc3;
    loc loc3: // put left fork
    do { left.isHeld := false; }
    goto loc0;
    loc loc4: // put left fork
    do { left.isHeld := false; }
    goto loc0;
    loc loc5: // put right fork
    do { right.isHeld := false; }
    goto loc0;
  }
}
```
A BIR Example – 2 Dining Philosophers

```java
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

```java
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;
}
```

Local variable declarations

```java
Demo
- Bogor BIR Editor
  - syntax highlighting
  - well-formed-ness checker
- Bogor Counter-example Display
  - states and transitions navigation
  - heap visualization
- Configuring Bogor
```

Outline

- Concept of Bogor
  - Extensions
    - Extending BIR language with new operations
    - Supporting Java
    - Functional sub-language
- Bogor Modeling Language and UI
  - Example: Dining philosophers
  - Demo: Bogor UI and BIR Case Wizard
- Conclusions
  - The utility of a customizable model-checking platform
**BIR: Extensible Modeling Language**

**Motivation**
- Variety of application domains and system level descriptions often work at different level of abstractions
  - want to be able to bridge the gap between system descriptions and BIR with ease
- BIR extensions...
  - can be extended on-demand
  - minimize changes and maximize reuse of Bogor components
    * parser/lexer, symbol table, AST, type system, etc.

**BIR Extensions**

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

  // declaration of abstract expressions
  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>);
}
```

**Sample usage**

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

---

**Domain-Specific Model-Checking**

**Bogor -- Extensible Modeling Language**

<table>
<thead>
<tr>
<th>Threads, Objects, Methods, Exceptions, etc.</th>
<th>Sets</th>
<th>Queues</th>
<th>RT CORBA Event Service API Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Modeling Language</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensions - Domain-specific Abstractions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bogor -- Customizable Checking Engine Modules</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State-space Exploration</td>
<td>State-space Representation</td>
<td>Quasi-cyclic Search</td>
<td>Partial State Representation</td>
</tr>
<tr>
<td>Core Checker Modules</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

**Extension Implementation**

Extensions are implemented by associating each item in extension interface with Java methods that provide the semantics for the item (or state-vector storage representation in case of state).

```plaintext
extension Set for SetModule {
  typedef type<'a>;
  expdef Set.type<'a> create<'a>('a ...);
  expdef 'a choose<'a>(Set.type<'a>);
  actiondef add<'a>(Set.type<'a>, 'a);
  expdef boolean forAll('a -> boolean, Set.type<'a>);
}
```

**Java implementation of set value and (state-vector) representation.**

**Java methods implementing actions and expressions.**

---

**Supporting Java**

- BIR provides features commonly found in modern programming languages
  - Dynamic creation of objects and threads, automatic memory management, etc.
- Java-to-BIR translator
  - Uses the Soot framework from Sable Research at McGill University
- Document: [http://projects.cis.ksu.edu/docman/?group_id=10](http://projects.cis.ksu.edu/docman/?group_id=10)

---

**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.)
- ...but only supports first-order functions
BIR Functional Sub-language

```java
record Node {
    Node next;
    int x;
}

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

Node for a linked-list data structure

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

Outline

- Concept of Bogor
- Extensions
  - Extending BIR language with new operations
  - Supporting Java
  - Functional sub-language

Coming Soon for Bogor...

- Sophisticated counterexample display facilities
- MSCs, abstractions of trace data, etc.
- Incorporation of a variety of forms of coverage information
- Support for a variety of forms of property specification/checking including...
  - Java Modeling Language (JML)
  - LTL/CTL via specification patterns
- Incorporation into next generation of Bandera
  ...many of these already implemented by not incorporated into distribution.

Conclusions

- General purpose state-space reduction strategies have dramatically reduced the cost of model-checking
  - yet it is still quite expensive to apply in many cases
- To obtain further significant reductions, we believe that a variety of domain knowledge can be leveraged to improve model-checking applicability
  - incorporating knowledge about domain specific data structures, scheduling policies, state invariants, etc.
- Bogor is a platform that is specifically designed to...
  - be used to obtain domain-specific model-checking engines
  - be used for exploring a variety of research directions related to model-checking

Bogor Modeling Language and UI

- Example: Dining philosophers
- Demo: Bogor UI and BIR Case Wizard

Conclusions

- The utility of a customizable model-checking platform
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 02: Core DFS Algorithm

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS04

Supplementary Material

- This material is part of a larger set of course notes on Foundations and Applications of Model-Checking
- Course notes are accompanied by a "Companion" that provides commentary and guided exercise solutions.

Outline

- BIR syntax and informal semantics
- Schedules and Computation Trees
- Simulating BIR systems using Bogor
- Verifying BIR systems using Bogor
  - Core DFS algorithm
  - Depth-bounded DFS search
  - Checking for invalid endstates (deadlock)

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.
- ...but we don’t need all of this to learn the basics of model-checking

Motivation and Goals

- One of our overall goals is to be able to customize and extend the guts of Bogor
- ...to do this, we need to understand the core model-checking algorithm
- Many, many aspects of model-checking tools and research (not just Bogor) build off the ideas in this lecture
- The goals of this lecture are to...
  - explain the core depth-first search algorithm used by Bogor (and many other model-checkers)
  - explain the primary data structures used by the DFS algorithm

BIR (simplified)

Subset of BIR

- Contains features that allow us to explore numerous issues related to modeling-checking in a simple and clean setting
- Does not contain the many features of BIR associated with OO software
  - omits methods, dynamically created data and threads, exceptions, etc.
- Has a straightforward correspondence with formal structures that are often used in the literature when describing foundational aspect of model-checking algorithms
- We will illustrate the syntax of BIR with an example
  - a grammar can be found in the appendix of the lecture notes.
A BIR execution can be viewed as a sequence of atomic steps or transitions between system states.

Intuitively, a BIR state holds all information that is necessary to carry on computation starting from a particular point in the system's execution.

A BIR state consists of...
- values of system variables
- current control location (program counter) of each thread

Notation for BIR State

System State

\[ [pc_1 \rightarrow 0, \text{fork}_1 \rightarrow false, \text{fork}_2 \rightarrow true] \]

...sometimes we will abbreviate as...
\[ [0, 1, false, true] \]

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

BIR Transition Notation

The thread Philosopher1 executes the transition leading out of loc2

A BIR State

BIR Execution Trace

An execution trace is a sequence of transitions between states

State Transition System

A state transition system is a mathematical structure that will allow us to reason precisely about the behavior of simple concurrent systems.

\[ \Sigma = (S, T, S_0, L) \]
...describes the behavior of a particular system

- S is a set of system states
- T is a set of transitions
- each transition is a relation between states
- \( S_0 \) is a set of initial states
- L maps a state to a set of primitive properties that are true of s (we won't use L for a while).
Initial States

**Σ_{2DP} initial states**
- BIR systems always have exactly one initial state
- Program counters for each thread are set to initial location (first location in thread code)
- Each BIR type has a default value
- In **Σ_{2DP}**, \( S_0 = [0, 0, false, false] \)

Transitions

**Σ_{2DP} transitions**
- For an arbitrary BIR system, there will be a transition in \( T \) for each guarded transformation in the BIR code.
- In **Σ_{2DP}**, there are eight transitions in \( T \) – one for each of the four transitions in each of the four threads.
- Each transition leading out of BIR location \( L \) in thread \( t \) has an implicit guard that only allows it to fire when \( t \)'s program counter is at \( L \).

Transitions as Relations/Functions

- Formally, \( \alpha \) is a subset of \( S \times S \), i.e.,
  - \( \alpha \) is a relation between states
  - Given an input state \( s \), \( \alpha(s) \) yields the states output by the transition.
- We will only need to consider deterministic transitions, i.e.,
  - Transitions that are (partial) functions, i.e.,
  - Transitions that, given an input state are undefined or produce exactly one output state.

Transition Examples

```plaintext
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;
}
```

Enabled/Disabled Transitions

- A BIR transformation is *enabled* for a particular state if both its explicit guard (on data variables) and implicit guard (on the current control position) are true.
- Relating to transitions from \( \Sigma \),
  - A transition \( \alpha \) is *enabled* for a state \( s \) if there exists a state \( s' \) such that \( \alpha(s) = s' \) (i.e., \( \alpha \) is defined on \( s \))
  - A transition \( \alpha \) is *disabled* for a state \( s \) if \( \alpha \) is undefined on \( s \).

Enabled/Disabled Notation

- \( enabled(s) \) – the set of transitions enabled at \( s \)
- \( enabled(s, t) \) – the set of transitions from thread \( t \) that are enabled at \( s \)
- \( disabled(s), disabled(s, t) \)
  - Similar to above
- \( pc(s, t) \) – the value in \( s \) of the program counter for \( t \)
- \( current(s), current(s, t) \)
  - The set of all transitions (whether enabled or disabled) associated with the current program counters of \( s \).
Enabled/Disabled Transitions

```
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;
}
```

Current Transitions

```
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;
}
```

Outline

- BIR syntax and informal semantics
- Schedules and Computation Trees
- Simulating BIR systems using Bogor
- Verifying BIR systems using Bogor
- Core DFS algorithm
- Depth-bounded DFS search
- Checking for invalid endstates (deadlock)

Objectives

- Understand the concept of thread interleaving in concurrent systems
- Understand how a system’s schedules can be viewed as a computation tree where a path through the tree corresponds to a particular schedule
- Be able to draw the computation tree for any simple BIR system

SumToN

```
system SumToN {
  const PARAM { N = 1; }
  typealias byte int wrap (0,255);
  byte x := 1;
  byte t1;
  byte t2;
  active thread Thread1() {
    loc loc0:
      when x != (byte)0 do { t1 := x; }
      goto loc1;
    loc loc1:
      do { t2 := x; }
      goto loc2;
    loc loc2:
      do { x := t1 + t2; }
      goto loc0;
  }
}
```

We use this example to motivate the concepts of schedule and computation tree.
SumToN

system SumToN {
const PARAM { N = 1 };  
typename byte int wrap (0,255);
byte x := 1;  
byte t1;  
byte t2;
active thread Thread1() {
loc loc0:  
when x != (byte)0 do { t1 := x; }  
goto loc1;
loc loc1:  
do { t2 := x; }  
goto loc2;
loc loc2:  
do { x := t1 + t2; }  
goto loc0;
}
active thread Thread2() {
loc loc0:  
when x != (byte)0 do { t1 := x; }  
goto loc1;
loc loc1:  
do { t2 := x; }  
goto loc2;
loc loc2:  
do { x := t1 + t2; }  
goto loc0;
}
active thread Thread0() {
loc loc0:  
do { assert (x != (byte)PARAM.N); }  
return;
}
}

The “main” thread asserts that x is not equal to the value of N.

Assessment

Can the assertion in the SumToN example be violated?
- Answering this question requires us to reason about possible schedules (i.e., orderings of instruction execution)
- Let’s try to find schedules that cause the assertion to be violated for various values of N...

Note: This transition can be arbitrarily interleaved with all others from Thread1 and Thread2.

Violating schedule for \( N = 1 \)

\[
\begin{align*}
(0,0,0,0,0,0,0) \rightarrow & \{0,0,0,0,1,1,0,1\} \rightarrow \{1,1,0,0,0,0,0,0\} \\
& \{0,0,0,0,0,0,0,0\}
\end{align*}
\]

...that was easy!
Computation Tree

- We can think of the possible schedules (execution traces) as forming a computation tree...

First example trace (schedule)

Second example trace (schedule)
Computation Tree

- We can think of the possible schedules (execution traces) as forming a computation tree...

For You To Do...

- Consider a modification to the SumToN system where N is set to 3. Is there a schedule that causes the assertion to be violated in this modified system? If so, give the schedule.
- For the SumToN system, does there exist a value of N from 0 and 255 such that assertion cannot be violated?
- Draw a computation tree five levels deep for the Bounded Buffer system from the lecture notes (i.e., the computation tree should represent the first five instructions of every possible schedule).

Summary

- The actions/ transitions of a concurrent system can be interleaved in arbitrary ways
- The implementation’s scheduler is responsible for choosing a particular ordering of transitions
- When verifying systems, one often abstracts away from the particular scheduling strategy and considers the choices between enabled transitions to be made non-deterministically
- The collection of execution traces of a system can be visualized as a computation tree
- A path through the computation tree corresponds to a particular execution (i.e., schedule).

Outline

- BIR syntax and informal semantics
- Schedules and Computation Trees
- Simulating BIR systems using Bogor
- Verifying BIR systems using Bogor
  - Core DFS algorithm
  - Depth-bounded DFS search
  - Checking for invalid endstates (deadlock)

Objectives

- Understand how a model-checker’s random simulation and user-guided simulation mode can be applied to explore paths through a system’s computation tree.
- Be able to apply Bogor in both random simulation mode and user-guided simulation mode.
Assessment

- Even though this is a very small system, it is already tedious for us to try to find a violating trace.
- Bogor can act as a simulator to help us find a violating trace.
- Bogor simulates in two ways:
  - random simulation
  - user-guided simulation
- Same as many other model-checkers (e.g., Spin)

Guided Simulation

- In a guided simulation, Bogor asks the user which transition to take at a choice point.

For You To Do...

- The computation tree for the SumToN example depicted on earlier slides is five levels deep. Extend the diagram to six levels.
- Download the file `sumton.bir` from the examples page.
- Run Bogor in random simulation mode.
  - Edit `sumton.bir` and change the assertion to `x != 1`.
  - Do you understand Bogor’s output? Was Bogor able to find a violating trace? Why/why not?
- Edit `sumton.bir` and change the assertion to `x != 3`.
- Run Bogor in random simulation mode several times. Do you understand Spin’s output? Was Bogor able to find a violating trace each time?
- Edit `sumton.bir` and change the assertion to `x != 5`.
- Using Bogor in guided simulation mode, construct a trace that leads to an assertion violation. Is this the shortest trace that leads to a violation? How can you be sure?
- Edit `sumton.bir` and change the assertion to `x != 7`.
- Using Bogor in guided simulation mode, construct a trace that leads to an assertion violation. Is this the shortest trace that leads to a violation? How can you be sure?

Assessment

- Bogor in random simulation mode...
  - sometimes it can find an error
    - e.g., when assertion read `x != 1`
  - most of the time it won’t find an error
    - e.g., when assertion read `x != 3`
  - in this case, Bogor runs forever, and you can notice the overflow error messages associated with the variables of type `byte`.

Random Simulation

- In a random simulation, Bogor randomly chooses a branch at a choice point.

Assessment

- Bogor in guided simulation mode...
  - the user can guide Bogor to the error
    - but this requires that the user already know or at least have a good idea about how the error can occur!
    - tedious and error prone
    - infeasible on all but very short traces
    - cannot be used in practice to obtain an exhaustive search of all possible traces
    - can be useful in practice if the user simply wants to explore the behavior, e.g., of a particularly troublesome section of code
On To Exhaustive Exploration...

- Bogor’s random simulation
  - isn’t that useful for finding bugs
  - only explores one execution trace
- Bogor’s 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 Bogor is its automatic exhaustive search capabilities
  - this is why people use model-checkers!
  - this is what this course is all about

Outline

- BIR syntax and informal semantics
- Schedules and Computation Trees
- Simulating BIR systems using Bogor
- Verifying BIR systems using Bogor
  - Core DFS algorithm
  - Depth-bounded DFS search
  - Checking for invalid endstates (deadlock)

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 systems.

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...
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.

Exhaustive Depth-first Search

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

... until the entire computation tree is covered.

For You To Do...

- Edit SumToN.bir and change the assertion to x != 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 Counterexample Display

Assessment

- Bogor spits out several types of 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).
- 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.

SumToN State Vector Example

- The state vector is the data structure corresponding to the state (as previously covered when discussing state transition systems for BIR). It holds the value of all variables as well as program counters for each process, and indicates a particular position in the computation tree.

Example State Vector: [0, 0, 2, 1, 1, 0]
The depth-first stack can be implemented to hold transitions
- requires less space, but ...(see next slide)...

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

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.
**Depth-first Stack of Transitions**

- 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.

**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).

**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.

**Revisiting Via A Different Path**

- This reduces amount of required memory and meshes well with its other space optimizations (e.g., bit-state hashing).

**Computation Tree as Graph**

- Sometimes we view the computation tree as a graph.

...sharing a node corresponds to revisiting a node that has been seen before.
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, 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.

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?

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

```
1    seen := {}    \text{current path being explored in computation tree begins with initial state}
2    stack := cons(so, null)    \text{current state to expand}
3
4    while stack != nil do \{    \text{remove s from current path}
5        s := head(stack) \text{if s has not been seen before, then}
6        stack := tail(stack) \text{get the transitions to explore at this state}
7        seen := seen union {s} \text{record that s has now been explored}
8        workSet := enabled(s) \text{pick one of the transitions to explore}
9        for all \( \alpha \in \text{workSet} \) \text{calculate the successor state}
10       s' := \alpha(s)    \text{put s' on the stack which represents}
11       stack := cons(s', stack) \text{the states which should still be expanded}
12
13    \}\text{remove s from current path}
14
15    if s has not been seen before, then \text{expand it}
16
17    \}\text{if s has not been seen before, then}
18
19 \}    \text{put s on the stack which represents}
20    \text{the states which should still be expanded}
```

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.
- The presentation on the previous page comes from the book of Godefroid's PhD thesis.

Outline

- BIR syntax and informal semantics
- Schedules and Computation Trees
- Simulating BIR systems using Bogor
- Verifying BIR systems using Bogor
  - Core DFS algorithm
  - Depth-bounded DFS search
  - Checking for invalid endstates (deadlock)

Outline

- Using Bogor's output to determine the max stack size (number of steps in longest execution path) encountered during the depth-first search
- Setting Bogor's depth bound
- Rationale for using depth-bounded search
- The consequences (unsoundness) of performing bounded searches
- Bogor options for finding minimal length counter-examples
- The core DFS algorithm extended to included depth-bounded search

Bogor Output

```
Running a model-check of SumToN with N = 5

Time: 4817 ms, Depth: 395 Error found: Assertion failed
Transitions: 38174, States: 15276, Matched States: 22899, Max Depth: 1921, Errors found: 19
Total memory before search: 329,240 bytes (0.31 Mb)
Total memory after search: 4,327,968 bytes (4.13 Mb)
Total search time: 4897 ms (0:0:4)
States count: 15276
Matched states count: 22899
Max depth: 1921
```

Error Trace Length

```
Model-check SumToN with N = 5
From Bogor's output, we can see that it has found a execution trace that violates the assertion and that the trace is 395 steps long.
- Having to reason about how the assertion can be violated along a trace of 395 steps is quite painful!
- You have previously discovered a much shorter violating trace using Bogor's simulation mode.
- Does this mean that the Bogor analyzer is not very useful?
  - Not at all!!
- We will see in a little bit how to tell Bogor to search for shorter violating traces (as well as minimal length violating traces).
```
Setting Bogor’s Depth Bound

- Users can set a bound on the depth of Bogor’s search (i.e., entries in Bogor’s depth-first stack).

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Bound</td>
<td>4</td>
</tr>
</tbody>
</table>

Choose the “Configure Bogor” option, then Add/Edit to set the value for the ISearcher.maxDepth attribute.

Setting Bogor’s Depth Bound

- This is often useful...
  - ...after a counterexample has been found and you want to see if a shorter one exists.
  - look at Bogor’s output to see the size, then rerun Bogor with an appropriate depth bound (i.e., one smaller than the size of the counter-example).
  - ...before a counterexample has been found and Bogor is taking too long or is running out of memory.

Bounded Depth-first Search

- When analyzing a system and given a depth bound as a command-line argument, Bogor will backtrack when that depth is reached.
Bounded Depth-first Search

- When analyzing a system and given a depth bound as a command-line argument, Bogor will backtrack when that depth is reached.

For You To Do...

- Edit `SumToN.bir` and change the value of `N` to 7.
- Use Bogor to find an error trace of minimal length.
  - start with a depth bound that allows an error
  - successively choose smaller versions of the bound until Bogor reports no error
  - determine a bound B such that running Bogor with bound B-1 reveals no errors, but running with B reveals an error
  - How does this error trace compare to the one (i.e., size and state vectors encountered) to the error trace that you discovered earlier using Bogor’s guided simulation mode?
  - Note that there may be multiple minimal length error traces.

Assessment

- Minimal length error traces can be found with the `ISearch.maxDepth` option for Bogor.
- This is somewhat tedious for the user.
- SPIN provides two other options to find shorter traces...
  - `pan.exe -i` finds a minimal length path by successively rerunning with bound set to length-of-current-violating-trace - 1 (can be costly!)
  - `pan.exe -l` similar to the option above but faster (a form of binary search is used), but approximate (sometimes minimal error trace is not found)
- We will eventually incorporate functionality similar to this in Bogor (it’s not difficult!)

Summary

- Bogor, like most other model-checkers, provides for a depth-bounded search
- A depth-bounded search can be useful when...
  - ...after a counterexample has been found and you want to see if a shorter one exists.
  - look at Bogor’s output to see the size, then rerun Bogor with an appropriate depth bound (i.e., one smaller than the size of the counter-example).
  - ...before a counterexample has been found and Bogor is taking too long or is running out of memory.
- However, a depth-bounded search is technically unsound since it gives rise to an under-approximation of the system’s behavior
  - In other words, an error may actually lie in an area that is outside of the area searched in the model-check
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

- Customizable Checking Engine Modules

A Bogor configuration is a key-value set

<table>
<thead>
<tr>
<th>Keys for component interfaces</th>
<th>Java class implementation for each interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>IActionTaker</td>
<td>DefaultActionTaker</td>
</tr>
<tr>
<td>IExpEvaluator</td>
<td>DefaultExpEvaluator</td>
</tr>
<tr>
<td>IEmbeddingStrategist</td>
<td>DefaultEmbeddingStrategist</td>
</tr>
<tr>
<td>IStateManager</td>
<td>DefaultStateManager</td>
</tr>
<tr>
<td>ITrafo</td>
<td>DefaultTrafo</td>
</tr>
<tr>
<td>IBackTrackingInfoFactory</td>
<td>DefaultBackTrackingInfoFactory</td>
</tr>
<tr>
<td>IStateFactory</td>
<td>DefaultStateFactory</td>
</tr>
<tr>
<td>IValueFactory</td>
<td>DefaultValueFactory</td>
</tr>
<tr>
<td>IStateManager</td>
<td>DefaultStateManager</td>
</tr>
<tr>
<td>ITransformer</td>
<td>DefaultTransformer</td>
</tr>
<tr>
<td>IActionTaker.setMaxErrors</td>
<td>1</td>
</tr>
</tbody>
</table>

...modular components with clean and well-designed API using design patterns
Bogor Initialization

Given a configuration, Bogor instantiates the specified components.

Options are passed to each component, and connections are established.

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

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

Package: bogor.type
Type and Value Representations

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

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

Outline

- Overview
  - Bogor components
  - Configuration
  - Initialization
- DFS Stack
  - Search algorithm
  - Scheduler
- Seen Before Set
  - State Manager

DefaultSearcher

A DFS implementation of the state-space exploration

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

while stack ≠ 0 do
  s := pop(stack)
  workSet := enabled(s)
  for each α ∈ workSet do
    s' := α(s)
    if s' ∈ seen then
      seen := seen ∪ {s'}
    end
    push(stack, s')
  end
end DFS
```

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

DefaultSearcher.step()

A DFS implementation of the state-space exploration

```
step()
{ If shouldBacktrack () \lor isSeen () \lor hasRunnableThreads () \lor isInvalidEndState () then
  return false
  if isInvalidEndState () then
    error(INVALID_END_STATE)
  return false
  T := ss.getEnabledTransformations (s)
  ssi := ss.getStrategyInfo (s)
  α := ss.advice(s, T, ssi)
  push(newBacktrackingInfo(s, T, α, ssi))
  doTransition(s, α, ssi)
  return true
  end
```
DefaultSearcher.step()

A DFS implementation of the state-space exploration

```java
void search() {
  ISchedulingStrategist ss;
  IState s;
  // ... code ...
}

public class DefaultSearcher
extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  void search() {
    s = createInitialState();
    void step() {
      // ... code ...
    }
  }
}
```

DefaultSearcher.backtrack()

A DFS implementation of the state-space exploration

```java
public class DefaultSearcher
extends ISearcher {
  IState s;
  ISchedulingStrategist ss;
  void search() {
    s = createInitialState();
    void step() {
      // ... code ...
    }
  }
}
```

Backtracking Information

- Used to keep track information for "undo"-ing transition
- Scheduling information for counter-example generation

- 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

- 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.

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

Backtracking Example

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

Scheduler

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

while stack ≠ ∅ do
  s := pop(stack)
  workSet := enabled(s)
  for each α ∈ workSet do
    s' := α(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

- 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
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 \textit{program counter} 
  - \( N \) is number of locations in the model
- Each variable is represented by its \textit{value}
  - for each ranged integer, \( N = \max - \min + 1 \)
  - value \( x \) is represented as \( x + \min \) using \( N \) bits
  - for enumeration, \( N \) is the number of enumeration elements
State Linearization

For the *TwoDiningPhilosophers* 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] ]

...similar bit patterns can be leveraged for reducing state-space representation (i.e., collapse compression)

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
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 04: Bogor Extensions

http://bogor.projects.cis.ksu.edu/
http://www.cis.ksu.edu/~hatcliff/ESSCaSS04

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 AIT
Rockwell-Collins ATC
Sun Microsystems

Customization Mechanisms

Bogor -- Extensible Modeling Language

Threads, Objects, Methods, Exceptions, etc. + Domain-Specific Abstractions

Core Modeling Language

Bogor -- Customizable Checking Engine Modules

Domain-Specific Scheduler
Domain-Specific Search
Domain-Specific State Rep.

Extensible Input Language – Resource Contention Example

how do we encode the representation in the model?

Imagine using an array to represent the resource pool

how do we represent the resource pool if we only care about whether a resource is acquired or not?

Domain-Specific Modeling

Bogor -- Extensible Modeling Language

Create new Bogor types and primitives

Java implementation of new values and new primitives inside model-checker

Imagine using an array to represent the resource pool

how do we represent the resource pool if we only care about whether a resource is acquired or not?
Bogor allows definitions of new abstract types and abstract operations as first-class constructs.

**Extension Implementation**
- Implementing the set value for the set type
  - operations on the value e.g., add, remove, isEmpty, etc.
  - visitor pattern for the value
  - used for linearization, etc.
  - linearization for state storage
  - fine-grained control over the value representation
  - XML externalization for counter-example display
Extension Implementation

Implementing the set value for the set type

```java
public interface ISetValue
    extends INonPrimitiveExtValue {
    void add(IValue v);
    boolean contains(IValue v);
    IValue[] elements();
    boolean isEmpty();
    void remove(IValue v);
}
```

Create an interface for all implementations of set values

Extension Implementation

Implementing the set value for the set type

```java
public class ReferenceElementSetValue
    implements ISetValue {
    protected HashSet set = new HashSet();
    public void add(IValue v) { set.add(v); }
    public boolean contains(IValue v) { return set.contains(v); }
    public boolean isEmpty() { return set.size() == 0; }
    public void remove(IValue v) { set.remove(v); }
    public IValue[] elements() {
        Object[] elements = set.toArray();
        orderValues(elements);
        IValue[] result = new IValue[elements.length];
        System.arraycopy(elements, 0, result, 0, elements.length);
        return result;
    }
}
```

Extension Implementation

Implementing the set value for the set type

```java
public class ReferenceElementSetValue
    implements ISetValue {
    protected HashSet set = new HashSet();
    public void add(IValue v) { set.add(v); }
    public boolean contains(IValue v) { return set.contains(v); }
    public boolean isEmpty() { return set.size() == 0; }
    public void remove(IValue v) { set.remove(v); }
    public IValue[] elements() {
        Object[] elements = set.toArray();
        orderValues(elements);
        IValue[] result = new IValue[elements.length];
        System.arraycopy(elements, 0, result, 0, elements.length);
        return result;
    }
    ...}
```

Extension Implementation

Implementing the visitor pattern for set values

```java
public class ReferenceElementSetValue
    implements ISetValue {
    public IValue[] elements() { ... }
    public void visit(...), boolean depthFirst, Set seen, LinkedList workList, ...) {
        IValue[] elements = elements();
        if (depthFirst) {
            for (int i = 0; i < elements.length; i++) {
                workList.addFirst(elements[i]);
            }
        } else {
            for (int i = 0; i < elements.length; i++) {
                workList.add(elements[i]);
            }
        }
    }
    ...
```

Extension Implementation

Implementing the visitor pattern for set values

Visitor pattern for traversing element values in the set (used for garbage collection, etc.)
Extension Implementation

Implementing the set value for the set type
- operations on the value
  - e.g., add, remove, isEmpty, etc.
- visitor pattern for the value
  - used for garbage collection, etc.
- linearization for state storage
  - fine-grained control over the value representation
- XML externalization for counter-example display

Extensible Input Language — Set Extension Example (Semantics)

The state of the set consists of encodings of the references to resources
Suppose the references to resources are represented as integers R, G, B
...convert to canonical order!

Extension Implementation

Implementing the XML externalization

An example XML representation of a set value with three elements: Disk#1, Disk#2, Display#1
Domain-Specific Modeling

Bogor -- Extensible Modeling Language

Core Modeling Language  New Bogor types and primitives  Java implementation of new values and new primitives inside model-checker

### Extension Implementation

**Implementing the set operations for the set type**

- **set module extension**
  - set creation
  - non-deterministically choose a set element
  - adding an element

```java
public class SetModule implements IModule {
    protected SymbolTable symbolTable;
    protected TypeFactory tf;
    protected IExpEvaluator ee;
    protected IValueFactory vf;
    protected ISchedulingStrategist ss;
    public String getCopyrightNotice() { return null; }
    public void setOptions(String key, Properties options) {}  
    public void connect(IBogorConfiguration bc) {
        symbolTable = bc.getSymbolTable();
        tf = symbolTable.getTypeFactory();
        ee = bc.getExpEvaluator();
        ss = bc.getSchedulingStrategist();
        vf = bc.getValueFactory();
    }
    public void dispose() {
        symbolTable = tf = ee = ss = vf = null;
    }
}
```

**Extension Implementation**

**Implementing the set operations for the set type**

- **Used for displaying copyright notices**

```java
public class SetModule implements IModule {
    protected SymbolTable symbolTable = bc.getSymbolTable();
    protected TypeFactory tf = symbolTable.getTypeFactory();
    protected IExpEvaluator ee = bc.getExpEvaluator();
    protected IValueFactory vf = bc.getValueFactory();
    protected ISchedulingStrategist ss = bc.getSchedulingStrategist();
    public String getCopyrightNotice() { return null; }
    public void setOptions(String key, Properties options) {}
    public void connect(IBogorConfiguration bc) {
        symbolTable = bc.getSymbolTable();
        tf = symbolTable.getTypeFactory();
        ee = bc.getExpEvaluator();
        ss = bc.getSchedulingStrategist();
        vf = bc.getValueFactory();
    }
    public void dispose() {
        symbolTable = tf = ee = ss = vf = null;
    }
}
```
public class SetModule implements IModule {
    protected SymbolTable symbolTable;
    protected TypeFactory tf;
    protected IExpEvaluator ee;
    protected IValueFactory vf;
    protected ISchedulingStrategist ss;
    public String getCopyrightNotice() { return null; }
    public void setOptions(String key, Properties options) {}
    public void connect(IBogorConfiguration bc) {
        symbolTable = bc.getSymbolTable();
        tf = symbolTable.getTypeFactory();
        ee = bc.getExpEvaluator();
        ss = bc.getSchedulingStrategist();
        vf = bc.getValueFactory();
    }
    public void dispose() {
        symbolTable = tf = ee = ss = vf = null;
    }
    ...
}
Extension Implementation

Implementing the set operations for the set type
- set module extension
- set creation
  - non-deterministically choose a set element
  - adding an element

ISchedulingStrategist

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

Extension Implementation

Implementing the set operations for the set type

public class SetModule implements IModule {
    // expdef 'a choose<'a>(Set.type<'a>);
    public IValue choose(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue[] elements = set.elements();
        int size = elements.length;
        int index = 0;
        if (size > 1) {
            index = ss.advise(arg.getExtDesc(),
                              arg.getNode(),
                              elements,
                              arg.getSchedulingStrategyInfo());
        }
        return elements[index];
    }
    ...
}

Extension Implementation

Implementing the set operations for the set type

public class SetModule implements IModule {
    // expdef 'a choose<'a>(Set.type<'a>);
    public IValue choose(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue[] elements = set.elements();
        int size = elements.length;
        int index = 0;
        if (size > 1) {
            index = ss.advise(arg.getExtDesc(),
                              arg.getNode(),
                              elements,
                              arg.getSchedulingStrategyInfo());
        }
        return elements[index];
    }
    ...
}

Extension Implementation

Get the set elements

Ask the scheduler which one to pick if there are two or more elements
Extension Implementation

Implementing the set operations for the set type

- Set module extension
- Set creation
- Non-deterministically choose a set element
- Adding an element

```
public class SetModule implements IModule {
    // actiondef add<'a>(Set.type<'a>, 'a);
    public IBacktrackingInfo[] add(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue element = (IValue) arg.getArgument(1);
        if (!set.contains(element)) {
            set.add(element);
            ISchedulingStrategyContext ssc =
                arg.getSchedulingStrategyContext();
            return new IBacktrackingInfo[] {
                createAddBacktrackingInfo(set, element, arg.getNode()
                    ssc.getStateId(), ssc.getThreadId(),
                    arg.getSchedulingStrategyInfo())
            };
        } else {
            return new IBacktrackingInfo[0];
        }
    }
}
```

Add the element if it is not already in the set value

```
public class SetModule implements IModule {
    // actiondef add<'a>(Set.type<'a>, 'a);
    public IBacktrackingInfo[] add(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue element = (IValue) arg.getArgument(1);
        if (!set.contains(element)) {
            set.add(element);
            ISchedulingStrategyContext ssc =
                arg.getSchedulingStrategyContext();
            return new IBacktrackingInfo[] {
                createAddBacktrackingInfo(set, element, arg.getNode()
                    ssc.getStateId(), ssc.getThreadId(),
                    arg.getSchedulingStrategyInfo())
            };
        } else {
            return new IBacktrackingInfo[0];
        }
    }
}
```

If the element is already in the set, then do nothing

```
public class SetModule implements IModule {
    // actiondef add<'a>(Set.type<'a>, 'a);
    public IBacktrackingInfo[] add(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue element = (IValue) arg.getArgument(1);
        if (!set.contains(element)) {
            set.add(element);
            ISchedulingStrategyContext ssc =
                arg.getSchedulingStrategyContext();
            return new IBacktrackingInfo[] {
                createAddBacktrackingInfo(set, element, arg.getNode()
                    ssc.getStateId(), ssc.getThreadId(),
                    arg.getSchedulingStrategyInfo())
            };
        } else {
            return new IBacktrackingInfo[0];
        }
    }
}
```

Backtrack by removing the element from the set

```
public class SetModule implements IModule {
    // actiondef add<'a>(Set.type<'a>, 'a);
    public IBacktrackingInfo[] add(IExtArguments arg) {
        ISetValue set = (ISetValue) arg.getArgument(0);
        IValue element = (IValue) arg.getArgument(1);
        if (!set.contains(element)) {
            set.add(element);
            ISchedulingStrategyContext ssc =
                arg.getSchedulingStrategyContext();
            return new IBacktrackingInfo[] {
                createAddBacktrackingInfo(set, element, arg.getNode()
                    ssc.getStateId(), ssc.getThreadId(),
                    arg.getSchedulingStrategyInfo())
            };
        } else {
            return new IBacktrackingInfo[0];
        }
    }
}
```
public class SetModule implements IModule {
    protected void addBacktrackingInfo(ISetValue set, final IValue element) {
        set.add(element);
    }

    public void backtrack(IState state) {
        set.remove(element);
    }

    public IBacktrackingInfo clone(Map cloneMap) {
        ISetValue clonedSet = (ISetValue) cloneMap.get(set);
        IValue clonedElement = (element instanceof INonPrimitiveValue) ?
            cloneMap.get(element) : element;
        return createAddBacktrackingInfo(cloneMap, clonedSet, clonedElement);
    }

    // Extension Implementation
    Extension Implementation

    Implementing the set operations for the set type

    (Deep) clone the backtracking information

    A BIR Example – Resource Contention

    Demo
    - Model with the Set extension
    - Incorporating the extension in the Bogor Eclipse plugin
    - Run the example
      - invalid end state
      - invariant checking

    Assessment

    - Bogor provides a clean and well-designed framework for extending its modeling language
    - Allows introduction of new abstract types and abstract operations as first-class construct
      - Complete control over value representation (linearization)
      - No double interpretation (the operations are executed as atomic actions in the model checker instead of being interpreted by the model checker)
    - Analogous to adding new instructions to a Virtual Machine
      - essentially, we are building abstract machines for particular domains
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 05: Representing Java in BIR

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS’04

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Matthew B. Dwyer
John Hatcliff
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

Slide Set 05: Representing Java in BIR

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Additional BIR identifiers

((|A|) [|this|]) …

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Records are used to model Java classes (inheritance)

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Static fields

static fields are modeled as global variables

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Records are used to model Java classes (inheritance)

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Static fields

static fields are modeled as global variables

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Records are used to model Java classes (inheritance)

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Static fields

static fields are modeled as global variables

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Records are used to model Java classes (inheritance)

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Virtual fields

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.

Modeling Java Programs

Static fields

static fields are modeled as global variables

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

Motivation and Goals
- BIR/Bogor provides direct support for representing Java (and e.g., C#)
- methods, exceptions, virtual method tables, references
- We will take a quick look at how Java is mapped to BIR.
Modeling Java Programs

Java methods as functions

```java
public class A {
    public static int N;
    public int x;
    protected int y;
    public A() {…}
    public void foo() {…}
}
```

class B extends A {
    public void foo() {…}
}

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

Java methods are modeled as functions

```java
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|) {
    function {|B.foo()|} ((|B|) [|this|]) …
}

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

Virtual tables are used to resolve dynamic dispatch of methods

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

BIR function invocations

```
function bar((|A|) a) {
    loc loc0: // invokespecial
    invoke {|A.foo()|} (a)
    goto loc0;
    loc loc1: // invokevirtual
    invoke virtual +|A.foo()|+ (a)
    goto loc1;
}
```

Dynamic dispatch of methods

```java
public class A {
    public static int N;
    public int x;
    protected int y;
    public A() {…}
    public void foo() {…}
}
```

class B extends A {
    public void foo() {…}
}

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

```java
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;
}
```

Exceptions

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

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

Throwables

```java
throwable t1;
```

```
function (|java.lang.Exception|) [t1] {
    (|java.lang.Throwable|) [t1];
    loc loc0: do { … } goto loc1;
    loc loc1: do { … } return;
    loc loc2: do { … } goto loc1;
}
```

Virtual tables are used to resolve dynamic dispatch of methods
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 06: Bogor’s Statespace Reductions

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS'04

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

Effective for OO Software

How do we take the well-known explicit-state model-checking algorithms and enhance them to be effective for working directly on software?

• How do we represent the state effectively?
• How do we reduce the number of paths/states explored?

Heap Representation

Simple Idea
• Start with something like Spin
• Implement objects as records/structs
• Implement heap as an array of objects

...not a good idea!

Heap Representations

Two Possible Schedules
1. R goes first 2. B goes first

Naïve Heap Representation

1. R goes first 2. B goes first

X
Y

Two Possible Schedules
1. R goes first 2. B goes first

Naïve Heap Representation

1. R goes first 2. B goes first

X
Y

Two Possible Schedules
1. R goes first 2. B goes first

Naïve Heap Representation

1. R goes first 2. B goes first

X
Y

Heap Representations

Thread R

1. R goes first

Thread B

2. B goes first

Two Possible Schedules

Naive Heap Representation

1. R goes first

2. B goes first

Observationally Equivalent

Bogor’s Heap Representation

Key Points...

- Explicit heap representation
- ...after each transition, a topological sort gives heap objects a canonical order
- Garbage is eliminated
- Precise heap model
- Precise alias information

Canonical heap

Canonical heap

... sort walks over heap, canonicalizes, and collects info

Heap Representation

- After each transition...
  - Reachable heap objects are ordered
  - Topological sort based on chains of field names
  - Unreachable objects are discarded (garbage collection)
- State compression...
  - Objects are held in a pool and are identified by bit patterns
  - State vector holds bit-vectors representing objects
  - Means that object values can be shared across states
    - Good! because in a typical transition, very little changes in the state
- Formalization of heap and thread symmetry...
  - Presentation by Radu Iosif based on group theory

Avoiding Equivalent Paths

Many paths are equivalent in the sense that they cannot be distinguished by the property being checked.

Partial Order Reduction (POR)

Properties of Independent Transitions

For each state \( s \in S \), and for each \( (\alpha, \beta) \in \mathcal{I}_t \):

- Preservation of Enablability: If \( \alpha, \beta \in \text{enable}(s) \) then \( \alpha, \beta \in \text{enable}(\beta(s)) \).
- Commutativity: If \( \alpha, \beta \in \text{enable}(s) \) then \( \beta(\alpha(s)) = \beta(\alpha) \).

Intuition

If a property to be checked, doesn’t make observations about \( \alpha, \beta \) then we only need to explore one of the paths.
**Classic PO Reductions**

- Usually based on syntactic inspection of the transitions (approximation)
  - e.g., accesses to local variables

```java
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;  // Traverse list of nodes
      head = head.next;
    }
  }
}
```

**Dynamic Object-Oriented Software**

Most data is heap-allocated, but it may still be local:

- Local data corresponds to **thread-local objects** - objects that are accessible by a single thread only.
- **Thread-local transitions** are transitions that do not access non-thread-local objects.
  - analogous to transitions that only access local variables
  - Thread-local transitions do not interfere with transitions from other threads - hence they should be considered independent.

```java
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;  // Traverse list of nodes
      head = head.next;
    }
  }
}
```

**Example**

Create a linked-list of heap allocated nodes

```java
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;  // Traverse list of nodes
      head = head.next;
    }
  }
}
```

How many of these transitions have to be interleaved with other threads?

**Example**

...none! they are all thread local!
**Static Approach**

- Static escape analysis can be used to detect thread-local objects (and thus, thread-local transitions).
- We implemented a modified version of Ruf’s escape analysis for Java.
- Concludes for the previous program that all transitions in the thread body do not access escaping objects.
- However, there exists many other opportunities for thread-local-based reductions in typical programs.
- Insight: an object can thread-local in some parts of the program but visible by more than one thread in others.

**Example Revisited**

```java
class Process extends Thread {
    Node head;
    public Process {
        head = new Node(0);
        Node temp = head;
        for (int i = 1; i < 10; i++) {
            temp.next = new Node(i);
            temp = temp.next;
        }
    }
    public void run() {
        while (head != null) {
            head.x++;
            head = head.next;
        }
    }
}
```

**Example Revisited -- Assessment**

Static escape analysis says these objects are escaping and, thus, are not thread-local.

**Limitations of Static Approach**

- To be classified as thread-local, conventional static escape analysis requires that an object \( o \) be accessible only by the same thread \( t \) throughout \( o \)'s entire lifetime.
- But note...
  - may have an object \( o \) that is thread-local for only part of its lifetime,
  - may have an object \( o \) that is thread-local to a thread \( t_1 \) for one part of its lifetime and thread-local to a thread \( t_2 \) for another part of its lifetime.
  - may have a transition that is thread-local on some executions but not on others.

**A Dynamic Framework**

- To allow flexible classification of objects...
  - we will carry out escape analysis dynamically (on-the-fly) and allow the classification of objects as thread-local to change over their lifetime.
- To allow flexible classification of transitions...
  - we will relax the conventional notion of independence relation to allow state-sensitive independence -- i.e., a pair of transitions can be independent in some states, but dependent in others.

**DFS(s)**

```java
DFS(s)
workSet := ample(s)
while workSet is not empty
    let \( \alpha \in workSet \)
    workSet := workSet - \{\alpha\}
    if not \( s' \in seen \) then
        seen := seen UNION \{s'\}
        pushStack(s')
        DFS(s')
        popStack()
end DFS
```

**Ample Sets POR Framework**

Main point: only explore transitions in ample set.

In our setting, we will construct ample(s) as follows:

- pick a thread \( t \) such that all its transitions at the current state \( s \) are thread-local, and let ample(s) be this set of transitions.
Assessment

- Replace a single fixed independence relation, \( I \), with a family of independence relations, \( I_s \), indexed by state \( s \)
  - This allows \((\alpha, \beta)\) to be independent in some states, but not in others.
- Correctness
  - Property preservation proofs for LTL-X
- Performance
  - Overhead for calculating \( I_s \) is small
  - Reduction is very large

Thread Locality: Performance

Example 1

Example 2

Leveraging Locking Information

Leveraging Locking Information

Leveraging Locking [Stoller '00]

Leveraging Locking Information

How Well Does It Work?

<table>
<thead>
<tr>
<th>Base Bogor + Syntactic Local Variable POR</th>
<th>Thread-local Reduction (20x smaller)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicated Workers</td>
<td>Thread-local Reduction (125x smaller)</td>
</tr>
<tr>
<td>Threads: 4</td>
<td>Threads: 4</td>
</tr>
<tr>
<td></td>
<td>Related</td>
</tr>
<tr>
<td>Location: 000</td>
<td>~10 hours</td>
</tr>
<tr>
<td>Time</td>
<td>14:49</td>
</tr>
<tr>
<td>Related</td>
<td>90407</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
</tbody>
</table>

Thread-local + Locking Reduction (125x smaller)

Thread-local + Locking + Read-only (1006x smaller)

>7000x reduction over old Spin-based version of Bandera!

How Well Does It Work?

Base Bogor + Syntactic Local Variable POR

Thread-local Reduction (20x smaller)

Thread-local + Locking Reduction (125x smaller)

Thread-local + Locking + Read-only (1006x smaller)
## How Hard To Implement?

- Dynamic Escape Analysis for POR
  - (150 LOC, 20x)
- Dynamic atomicity
  - (5 LOC, 5x)
- Extension of Stoller's Locking-discipline
  - (100 LOC, 20x)

Researchers at NASA Ames have now incorporated these reduction strategies into their JPF model-checker.

See "Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs". FMSD 2004
Software Model Checking Using Bogor
- a Modular and Extensible Model Checking Framework

3rd Estonian Summer School in Computer and System Science (ESSCaSS04)

Slide Set 07: Checking JML Specifications

http://bogor.projects.cis.ksu.edu/
http://www.cs.ksu.edu/~hatcliff/ESSCaSS04

John Hatcliff Matthew B. Dwyer Robby
SrnToS Laboratory, Kansas State University, USA

Motivation and Acknowledgements

- All other model-checkers that we know of support only simple predicates on system states (e.g., the primitive propositions occurring in temporal logic formulas).
- Especially when modeling OO languages, states themselves can be quite complicated (they include “live heap”).
- Therefore we are interested in supporting specification predicates over Bogor states that are significantly stronger than those supported in other model-checking frameworks.
- Moreover, we are interested in supporting, as much as possible, rich specification languages that other verification tools using different technologies (e.g., theorem proving) also support.
- These slides are taken from our talk given at TACAS 2004 on “Checking Strong Specifications Using an Extensible Model-Checking Framework”
- A significant portion of this work was carried out by Edwin Rodriguez

Assertions for Software Verification

- Assertions have become a common practice among developers
- 10 years ago assertions were not considered useful by developers
- Recent evidence of the effectiveness of assertions
  - David Rosenblum (1995)
  - Now some programming languages have included assertions in their standard specifications
  - C.f. Java 1.4 assertions

An example

```java
public class LinkedQueue {
    protected final Object putLock;
    public class LinkedNode {
        public Object value;
        public LinkedNode next;
    }
    public void put(Object x) {
        synchronized (head) {
            if (x != null) return x;
            if (x == null)
                throw new IllegalArgumentException();
            assert(putLock != null);
        }
    }
}
```

Specify that putLock is never null
Specification Languages

- We want specification languages that
  - have a rich set of primitives for observing program state
    - heap-allocated objects, concurrency, etc.
  - make it easy to write useful specifications
  - support lightweight and deep-semantic specifications
  - be checkable using a variety of analysis techniques
    - static analysis, theorem proving, etc.

Java Modeling Language (JML)

- Developed by G. Leavens and other colleagues at Iowa State University
  - very rich set of operators, especially for describing complex heap properties
    - \( \text{reach}(r), \forall() \), \( \text{old}() \), etc.
  - support for specifications with varying degrees of complexity
    - lightweight vs. heavyweight specifications
  - has been checked with a variety of different techniques
    - so far, static analysis, theorem proving and runtime checking
  - Emerging as a standard specification language for Java within the research community

Java Modeling Language (JML)

```
public class LinkedQueue {
    protected final /*@ non_null @*/ Object putLock;
}

protected synchronized Object extract() {
    synchronized (putLock) {
        Object x = null;
        if (first != null){
            first.value = null;
            x = first.value;
            if(waitingForTake > 0) putLock.notify();
            if(x != null) return x;
        } else {
            return;
        }
        return x;
    }
}
```

```
protected void insert(Object x) {
    synchronized (putLock) {
        Object x = null;
        if(x != null) return x;
    }
    synchronized (putLock) {
        Object x = null;
        if(x != null) return x;
    }
    protected synchronized void insert(Object x) {
        assert(x != null);
        synchronized (putLock) {
            LinkedNode p = new LinkedNode(x);
            synchronized (last) {
                last = p;
                if(waitingForTake > 0) putLock.notify();
            }
        }
    }
}
```

```
public class LinkedNode {
    protected int waitingForTake = 0;
    protected LinkedNode last = head;
    protected LinkedNode head;
    protected final Object putLock;
    protected void insert(Object x) {
        assert(putLock != null);
        synchronized (putLock) {
            LinkedNode p = new LinkedNode(x);
            last = p;
            if(waitingForTake > 0) putLock.notify();
        }
    }
    public LinkedNode(Object x) {
        assert(putLock != null);
        synchronized (putLock) {
            LinkedNode p = new LinkedNode(x);
            last = p;
            if(waitingForTake > 0) putLock.notify();
        }
    }
}
```

An example

```
public class LinkedQueue {
    protected final /*@ non_null @*/ Object putLock;
    protected synchronized Object extract() {
        Object x = null;
        if (first != null){
            first.value = null;
            x = first.value;
            if(waitingForTake > 0) putLock.notify();
            if(x != null) return x;
        } else {
            return;
        }
        return x;
    }
}
```

Tool support for JML

- Many tools have been developed to support verification of JML
  - jmic (Leavens et al)
  - LOOP (Jacobs et al)
  - ESC/Java (Compaq SRC)
  - KeY (Ahrendt et al)
  - Calvín (Flanagan et al)
  - JACK (Burdy et al)
  - ChAsE (N. Catao)
  - Krakatoa (Marché et al)
  - jive (Poetzsch-Henrich et al)

- Every tool provide different trade offs in terms of several factors
  - coverage of the JML language
  - coverage of Java
  - degree of automation
  - scalability
Assessment of JML Tools

<table>
<thead>
<tr>
<th>JML coverage</th>
<th>Behavior coverage</th>
<th>Degree of Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>moderate</td>
<td>moderate</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>low</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>JACK</td>
<td>JACK</td>
<td></td>
</tr>
<tr>
<td>jmlc</td>
<td>jmlc</td>
<td></td>
</tr>
<tr>
<td>ChAsE</td>
<td>ChAsE</td>
<td></td>
</tr>
<tr>
<td>ESC/Java</td>
<td>ESC/Java</td>
<td></td>
</tr>
</tbody>
</table>

Motivation: explore model checking as a technique to fill these gaps

Questions...
- What is it?
- Why is it useful?
- What makes it particularly good for checking JML?

Bogor's Heap Representation

Key Points...
- ...transition may create new objects, garbage, etc.
- ...transition may create new objects, garbage, etc.
- ...sort walks over heap, canonicalizes, and collects info
- ...sort walks over heap, canonicalizes, and collects info
- ...precise heap model
- ...precise alias information
- ...have access to all visited states (but, efficiently stored using collapse compression)
- ...can easily compare objects in methods pre/post-states (c.f., |old|)

Bogor's Heap Representation – Enables JML Specs Check

Key Points...
- ...many JML features are easy to support in Bogor
- ...precise heap model (c.f., |reach|)
- ...precise alias information (c.f., assignable)
- ...can easily compare objects in methods pre/post-states (c.f., |old|)
Checking JML Specs with Bogor

- Object operations
  - `assignable`, `\reach(x)`, `\lockset`, `\fresh(x_1, ..., x_n)`
  - Quantification over objects of a specified type
    - `forall()`, `\exists()`
  - Pre-/post-conditions, invariants
  - Referencing Pre-states
  - Methods in JML expressions (the purity issue)

Object operations - `assignable`

- `assignable` allows to specify frame conditions for a method
  - `assignable v_1, v_2, ..., v_n;`
    - `v_1, v_2, ..., v_n` can be modified by the method
    - Modification to any other memory location is forbidden
  - Traditionally hard to check
    - Aliasing makes it hard to determine unambiguously which memory locations can actually be assigned to
    - Verified conservatively in the best cases

Object operations - `assignable`

- Maintaining a precise, dynamic heap model allows performing accurate object operations
  - Eliminated aliasing issues when checking `assignable`
  - Other object operations easily performed too
    - `\reach(x)` - simple DFS traversal from `x`
    - `forall` - compute quantification set by DFS from root objects, then post-filtering by type

Object operations

- Precision alias information... exact assignability verification!

LinkedQueue Example (JML)

```java
public class LinkedQueue {
    protected final /*@ non_null @*/ LinkedNode head;
    protected /*@ non_null @*/ LinkedNode last = head;
    protected final /*@ non_null @*/ Object putLock;
    protected int waitingForTake = 0;
    protected /*@ non_null @*/ LinkedNode last = head;
    protected int waitingForTake = 0;
    protected synchronized Object extract() {
        return head.next == null;
    }
    public void m() {
        @ assignable a.f;
        @ assignable last, last.next;
        @ assignable a, v1, v2, ..., vn;
        @ assignable v_1, v_2, ..., v_n;
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
        @ assignable
```

Maintaining a precise, dynamic heap model allows performing accurate object operations
- Eliminated aliasing issues when checking `assignable`
- Other object operations easily performed too
  - `\reach(x)` - simple DFS traversal from `x`
  - `forall` - compute quantification set by DFS from root objects, then post-filtering by type
public class LinkedNode {
    public Object value;
    public LinkedNode next;
    @behavior
    @ ensures value == x;
    @*/
    public LinkedNode(Object x) {
        value = x;
    }
}

class LinkedQueue {
    protected final /*@ non_null @*/ Object putLock;
    protected /*@ non_null @*/ LinkedNode head;
    protected /*@ non_null @*/ LinkedNode last = head;
    protected int waitingForTake = 0;
    @instance invariant waitingForTake >= 0;
    @*/
    public LinkedQueue() {
        putLock = new Object();
        head = new LinkedNode(null);
    }
    @behavior
    @ assignable head, last, putLock, waitingForTake;
    @ ensures \fresh(head, putLock) && head.next == null;
    @*/
    public boolean isEmpty() {
        synchronized (head) {
            return head.next == null;
        }
    }
    @behavior
    @ requires n! = null;
    @ assignable last, last.next;
    @*/
    protected void refactoredInsert(LinkedNode n) {
        last.next = n;
        last = n;
    }
    @behavior
    @ requires x! = null;
    @ ensures true;
    @ also behavior
    @ requires x= = null;
    @ signals (Exception e) e instanceof IllegalArgumentException;
    @*/
    public void put(Object x) {
        if (x == null)
            throw new IllegalArgumentException();
        insert(x);
    }
    protected synchronized Object extract() {
        synchronized (head) {
            return refactoredExtract();
        }
    }
    @behavior
    @ assignable head, head.next.value;
    @ ensures \result == null || ( \exists LinkedNode n;
        @ \old(\reach(head)).has(n);
        @ n.value == \result
        @& & ! ( \reach(head).has(n)));
    @*/
    protected Object refactoredExtract() {
        Object x = null;
        LinkedNode first = head.next;
        if (first != null){
            x = first.value;
            first.value =null;
            head = first;
        }
        return x;
    }
    @behavior
    @ requires x! = null;
    @ ensures last.value == x && \fresh(last);
    @*/
    protected void insert(Object x) {
        synchronized (putLock) {
            LinkedNode p = new LinkedNode(x);
            synchronized (last) refactoredInsert(p);
            if (waitingForTake > 0) putLock.notify();
        }
    }
}

LinkedQueue Example (JML)

/*@ behavior
@ requires x != null;
@ ensures last.value == x && \fresh(last);
@*/
protected void insert(Object x) {
    synchronized (putLock) {
        LinkedNode p = new LinkedNode(x);
        synchronized (last) refactoredInsert(p);
        if (waitingForTake > 0) putLock.notify();
    }
}

Pre/Post-Conditions

// jmlc generates a wrapper method for each method in the class

Pre/Post-Conditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

At this point a thread can interleave and insert an object in the LinkedQueue; so there actually exists an execution race where the post-condition is violated.
Pre/Post-Conditions

- Object operations
  - assignable, \( \text{reach}(x), \text{lockset}, \text{fresh}(x_1, \ldots, x_n) \)
- Quantification over objects of a specified type
- Pre-/post-conditions, invariants
- Referencing Pre-states
- Methods in JML expressions (the purity issue)

Checking JML Specs with Bogor

JML’s \( \text{old()} \) clause

- The \( \text{old()} \) clause provides a way to access pre-state values in post-state conditions
  - e.g. ensures \( \text{old}(a) + 1 == a; \)
  - asserts that the current value of \( a \) has increased by one w.r.t. the value that it had at the beginning of this method
- very useful for constraining the behavior of a method

Issues with \( \text{old()} \) and model checking

/*@ behavior */
/*@ ensures \text{old}(a) + 1 == a; */
public void m() {
    a = 1;
    b = 1;
}

Issues with \( \text{old()} \) and model checking

/*@ behavior */
/*@ ensures \text{old}(a) + 1 == a; */
int cs = collapsedState(a);
public void m() {
    a = 1;
    b = 1;
}
Methods in JML expressions

- All methods called from JML expressions must be pure:
  - A pure method must be guaranteed to have no side effects
  - ... must refine assignable \nothing;
  - JML considers the locking used in synchronization as a kind of side effect
  - synchronized methods cannot be declared pure

The purity issue

- Runtime checking
  - different behavior than without runtime checking

- Model checking
  - ... allows to define a kind of weak purity
public class LinkedNode {
    public Object value;
    public LinkedNode next;
    @<@ behavior
    @<@ ensures value == x;
    @<@
    public LinkedNode(Object x) {
        value = x;
    }
}

public class LinkedQueue {
    protected final /*@ non_null @*/ Object putLock;
    protected /*@ non_null @*/ LinkedNode head;
    protected /*@ non_null @*/ LinkedNode last = head;
    protected int waitingForTake = 0;
    @<@ instance invariant waitingForTake >= 0;
    @<@ instance invariant \reach(head).has(last);
    @<@ behavior
    @<@ assignable head, last, putLock, waitingForTake;
    @<@ ensures \fresh(head, putLock) && head.next == null;
    @<@
    public LinkedQueue() {
        putLock = new Object();
        head = new LinkedNode(null);
    }
    @<@ behavior
    @<@ ensures \result <==> head.next == null;
    @<@
    public boolean isEmpty() {
        synchronized (head) {
            return head.next == null;
        }
    }
    @<@ behavior
    @<@ requires n! = null;
    @<@ assignable last, last.next;
    @<@ ensures last.value == x && \fresh(last);
    @<@
    protected void refactoredInsert(LinkedNode n) {
        last.next = n;
        last = n;
    }
    @<@ behavior
    @<@ requires x! = null;
    @<@ ensures true;
    @<@ also behavior
    @<@ requires x== null;
    @<@ signals (Exception e) e instanceof IllegalArgumentException;
    @<@
    public void put(Object x) {
        if (x == null)
            throw new IllegalArgumentException();
        insert(x);
    }
    @<@ behavior
    @<@ assignable head, head.next.value;
    @<@ ensures \result == null || (\exists LinkedNode n; @<@ \old(\reach(head)).has(n);
    @<@ n.value == \result
    @<@ & & ! ( \reach(head).has(n)));
    @<@
    protected Object refactoredExtract() {
        Object x = null;
        LinkedNode first = head.next;
        if (first != null) {
            x = first.value;
            first.value = null;
            head = first;
        }
        return x;
    }
    @<@ behavior
    @<@ requires x! = null;
    @<@ ensures last.value == x && \fresh(last);
    @<@
    protected void insert(Object x) {
        synchronized (putLock) {
            LinkedNode p = new LinkedNode(x);
            synchronized (last) refactoredInsert(p);
            if (waitingForTake > 0) putLock.notify();
        }
    }
}

LinkedQueue Example (JML)

/*@ behavior
@ ensures \result <==> head.next == null;
@*/
public boolean isEmpty() {
    synchronized (head) {
        return head.next == null;
    }
}

LinkedQueue Example
LinkedQueue Example

```java
public class LinkedQueue{
    public static void main(String[] args) {
        // Create a LinkedQueue instance
        LinkedQueue queue = new LinkedQueue();

        // Enqueue elements
        queue.enqueue(10)
        queue.enqueue(20)
        queue.enqueue(30)

        // Dequeue elements
        System.out.println(queue.dequeue()); // 10
        System.out.println(queue.dequeue()); // 20
        System.out.println(queue.dequeue()); // 30
    }
}
```

LinkedQueue Example

```java
public class LinkedQueue{
    public static void main(String[] args) {
        // Create a LinkedQueue instance
        LinkedQueue queue = new LinkedQueue();

        // Enqueue elements
        queue.enqueue(10)
        queue.enqueue(20)
        queue.enqueue(30)

        // Dequeue elements
        System.out.println(queue.dequeue()); // 10
        System.out.println(queue.dequeue()); // 20
        System.out.println(queue.dequeue()); // 30
    }
}
```

JML Language Coverage

- assignable
- invariant
- valid
- assignableInv
- constrainable
- ensures
- requires
- ensuresInv
- constrainableInv
- nonnull
- nonnullInvariance
- nonnullInv
- nonnullInvarianceInv
- set
- setConstraction
- mustBe
- mayBe
- assignableRef
- type

large language coverage...

Preliminary Results

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>w/ JML</th>
<th>w/o JML</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedQueue()</td>
<td>296 sec</td>
<td>57 sec</td>
</tr>
<tr>
<td>3 objects</td>
<td>39 sec</td>
<td>21 sec</td>
</tr>
<tr>
<td>5 objects</td>
<td>35 sec</td>
<td>14 sec</td>
</tr>
<tr>
<td>20 objects</td>
<td>245 sec</td>
<td>125 sec</td>
</tr>
</tbody>
</table>

ROM (12) 227 sec

- 1 object
- 3 objects
- 5 objects
- 7 objects
- 9 objects
- 11 objects
- 13 objects
- 15 objects
- 17 objects
- 19 objects
- 21 objects

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>w/ JML</th>
<th>w/o JML</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedQueue()</td>
<td>296 sec</td>
<td>57 sec</td>
</tr>
<tr>
<td>3 objects</td>
<td>39 sec</td>
<td>21 sec</td>
</tr>
<tr>
<td>5 objects</td>
<td>35 sec</td>
<td>14 sec</td>
</tr>
<tr>
<td>20 objects</td>
<td>245 sec</td>
<td>125 sec</td>
</tr>
</tbody>
</table>

ROM (12) 227 sec
Bogor’s Reduction Algorithms — Enables Checking JML Specs

Indicates little overhead compared with simply exploring the state-space

Assessment of JML Tools

Behavior coverage

SpEx-JML

ChAsE

ESC/Java

JML coverage

Degree of Automation

JML Eclipse

JML annotated

Java source

JML syntax highlighting

JML well-formedness checking

JML Eclipse

JML well-formedness checker
## Future Work

- Proposal for concurrency specifications in JML
  - thread-locality
  - regionized pre-/post-conditions
  - atomicity, etc.
- JMLEclipse as an open ended JML plugin for Eclipse
- Other specification formalisms

## For More Information...

- SAnToS Laboratory, Kansas State University
  [http://www.cis.ksu.edu/santos](http://www.cis.ksu.edu/santos)
- Bogor Project
  [http://bogor.projects.cis.ksu.edu](http://bogor.projects.cis.ksu.edu)
- SpEx Project
  [http://spex.projects.cis.ksu.edu](http://spex.projects.cis.ksu.edu)
- JMLEclipse Project
  [http://jmleclipse.projects.cis.ksu.edu](http://jmleclipse.projects.cis.ksu.edu)
- Bandera Project
  [http://bandera.projects.cis.ksu.edu](http://bandera.projects.cis.ksu.edu)
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 08: Cadena Overview

http://bogor.projects.cis.ksu.edu/
http://www.cis.ksu.edu/~hatcliff/ESSCaSS04

John Hatchfield 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

Acknowledgements

Work on Cadena has been carried out by the following team of people
- PIs: John Hatchfield, Matt Dwyer, Gurdip Singh
- Primary Developers: Jesse Greenwald, Venkatesh Ranganath, Adam Childs, Prashant Kumar Shanti
- Students: Georg Jung, William Deng, Matt Hoosier

Goals of the Cadena Project

I. Platform for real-world experimentation with technologies for building high-assurance distributed systems using CORBA Component Model

- light-weight specification, analysis, and verification techniques
- model-based development, middleware configuration, and code synthesis

II. Avenue for collaborating with industrial research teams and middleware experts to guide next-generation component/middleware technology

- interacting with groups at Boeing, Rockwell Collins, Lockheed Martin to develop techniques that match fit into development process
- collaborating with middleware experts (e.g., ACE/TAO RT Middleware) to make frameworks more amenable to model-based configuration and analysis

Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
  - component development
  - lightweight semantic annotations
    - intra-component dependences
    - intra-component transition semantics
  - system assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor’s modeling language to support Cadena designs
- Customizing Bogor’s scheduling and state-space search modules to Cadena/BoldStroke designs

Distributed Components

An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-based Systems

At the heart of Cadena is an integrated development environment for analysis, synthesis, and verification of component-based systems. This environment is designed to support a robust tool environment suitable for industrial experimentation and model-based development, middleware configuration, and code synthesis.

Distributed Components

The diagram illustrates the distributed components of a network system, showcasing the interconnectivity of various programming languages and middleware technologies. This visualization helps in understanding the architecture and interactions within a distributed computing environment.
Objects to Components

- Consider: group of objects working together to provide a service to clients.
- Objects are meant to be used "as a team".
- No language mechanism to identify components as a single group.
- Harder for 3rd parties to reuse and assemble.

Components collect related classes together to form a coarser-grain composable unit.
- Components explicitly define interfaces they provide to their clients.
- Components indicate the other interfaces/events they depend on.
- Considerable auto-coding functionality provided.

Checking CCM Systems

Modern Software Systems

- These systems are huge!
- Extensive use of OO patterns & software layering
- What are appropriate abstractions for formal reasoning?
- How can we help developers write them?
- Useful properties?
- How must conventional model-checking engines be extended?

Component-based Design

- Development of component development environments allows model-based development of Bold Stroke applications using the CORBA Component Model (CCM).
- Automatic generation of component infrastructure code using CCM IDL compilers.
- Development of core functional code (business logic) using Eclipse Java facilities.

Leverage CORBA IDL

- automatic code generation
- Component infrastructure implementation
- Core functional code written by component developer

Components require an interface and provide an interface.
- Component Development
- Interface Ports
- Input event port
- Output event port
- Core functional code written by component developer
- Component Implementation Stubs & Skeletons
- IDL Compiler
- Dependency Analysis and Model-checking Engine
**Incremental Specification**

- **Specifications**
  - port action dependencies
  - state-based dependencies
  - component transition semantics

- **Component Structure**
  - Increasing Effort & Strength of Verification
  - ...only in mode Y
  - ...state machines give abstract behavior

**Component Integration**

Multiple views for allocating component instances and connecting components together to form a system assembly.

**Model-based Programming**

Programming at a higher level of abstraction...

Various analyses guide system development...

Many system elements – configuration of communication services, setting of QoS properties, etc. – are programmed by selecting particular attribute values at the model level.

**Model-level Analysis**

Analysis facilities provide multiple forms of a design-level slicing, chopping, etc. and model-checking of global temporal properties.

**Packaging & Deployment**

XML-based Configuration and Deployment Information

CCM Deployment Infrastructure

**Lecture Outline**

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
  - component development
  - lightweight semantic annotations
  - intra-component dependences
  - intra-component transition semantics
  - system assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor’s modeling language to support Cadena designs
- Customizing Bogor’s scheduling and state-space search modules to Cadena/BoldStroke designs
Avionics Mission Control Systems

- Mission-control software for Boeing military aircraft
- Boeing’s Bold Stroke Avionics Middleware
  - built on top of ACE/TAO RT CORBA
- Provided with an Open Experimental Platform (OEP) from Boeing
  - a sanitized version of the real system
  - 100,000+ lines of C++ code (including RT CORBA middleware)
  - 50+ page document that outline development process and describe challenge problems
- Must provide:
  - tool-based solutions that can be applied by Boeing research team to realistic systems
  - solutions that fit within current development process, code base, etc.
  - metrics for that allow Boeing research team to evaluate tool performance and ease of use

Control-Push Data-Pull

Typical situation

Component A computes some data that is to be read by one or more components B_i.

Run-time Actions

- A publish a dataAvailable event
- B_i call the getData() method of A to fetch the data

Control-Push Data-Pull Structure

1. Logical GPS component receives a periodic event indicating that it should read the physical GPS device.
2. Logical GPS publishes DATA_AVAILABLE event
3. Airframe component fetches GPS data by calling GPS GetData method
4. Airframe updates its position data and publishes DATA_AVAILABLE event
5. NavDisplay component fetches AirFrame data by calling AirFrame GetData method
6. NavDisplay updates the physical display

Larger Configuration

...moving up to 1000+ components

System Requirements

Input Requirements

- The system shall request new inputs from the GPS subsystem at a 40 Hz rate.
- The system shall poll for a pilot steering mode input at a 1 Hz rate.
- The system shall receive data from the navigator controls at a 5 Hz rate.

Output Requirements

- The system shall disable the display of steering information when deselected by the pilot.
- When the navigation steering mode is selected, the system shall:
  - Update navigation steering information display outputs at 20 Hz rate based on current airframe data and the current list of navigation points that have been submitted by the navigator. The latency between the GPS inputs and the display output shall be less than a single 20 Hz frame. The latency between navigation point input and the associated output shall be less than a single 5 Hz frame.
  - When the tactical steering mode is selected, the system shall:
    - Update tactical steering information display outputs whenever the aircraft position data changes.
    - The system shall display new aircraft position data at a 20 Hz rate. The latency between associated inputs and this output shall be less than a single 20 Hz frame.

System Design Aspects

Map components to onboard network nodes

Implement mode semantics for changing subsystem behavior
Development Process

Component Development
- Common Components
- Platform-specific Components

Component Integration
- Connect components, assign priorities, locking schemes, distribute

Real Board Testing
- Test real-time aspects, frame-overruns, etc.

Analysis & Functional Testing
- Debuggers, call-graph analyzers, scheduling tools

Lack of Model Analysis

Boeing OEP Challenge Problems
1. Forward & backward data and event dependencies
2. Dependency intersections
3. Components with high data coupling
4. All components from a particular rate group
5. Cycle checks
...15-20 others related to dependencies

A temporal property well-suited for model-checking!

Lecture Outline
- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
  - component development
  - lightweight semantic annotations
  - intra-component dependences
  - intra-component transition semantics
  - system assembly
  - Analysis, automated design device, analysis driven configuration and customization of middleware and services
  - Extending Bogor’s modeling language to support Cadena designs
  - Customizing Bogor’s scheduling and state-space search modules to Cadena/BoldStroke designs

No Unifying Mechanism

C++ Component Code
- Configurator Info
- Integrated Development Environment

High-level Specification Language
- Design Artifacts
- Analysis and QoS Aspect Synthesis

Cadena - CCM Development

High-level specification of abstract component behavior
- Visualization and design-level reasoning
- Eclipse Plug-In
- Integrated Development Environment
- Code generation functions (via OpenCCM) produces code amenable to conformance checking and certification
Component IDL

```idl
component BMLazyActive {
    provides ReadData outData;
    uses ReadData inData;
    publishes DataAvailable outDataAvailable;
    consumes DataAvailable inDataAvailable;
    attribute LazyActiveMode dataStatus;
};
```

Leverage CORBA IDL

```idl
component BMLazyActive {
    provides ReadData outData;
    uses ReadData inData;
    publishes DataAvailable outDataAvailable;
    consumes DataAvailable inDataAvailable;
    attribute LazyActiveMode dataStatus;
};
```

Incremental Specification

- Specifications
  - port action dependencies
  - refinement
- Component Structure
  - state-based dependencies
  - refinement
  - component transition semantics

```
dependencydefault == none;
dependencies {
    dataWriteOut.set_data() -> outDataAvailable;
}
```

```
dependencydefault == all;
dependencies {
    modeChange() ->;
    case modeChange.modeVar of {
        enabled: inDataAvailable -> dataIn.get_data(),
        outDataAvailable;
        disabled: inDataAvailable ->;
    }
}
```

Outline

2. Dependence Information

```
dependencydefault == none;
dependencies {
    case modeChange.modeVar of {
        enabled: isDataAvailable -> dataIn.set_data(),
        outDataAvailable;
        disabled: isDataAvailable ->;
    }
}
```

Light-weight Dependency Specs

```
dependencydefault == none;
dependencies {
    case modeChange.modeVar of {
        enabled: isDataAvailable -> dataIn.set_data(),
        outDataAvailable;
        disabled: isDataAvailable ->;
    }
}
```

```
dependencydefault == all;
dependencies {
    modeChange() ->;
    case modeChange.modeVar of {
        enabled: isDataAvailable -> dataIn.set_data(),
        outDataAvailable;
        disabled: isDataAvailable ->;
    }
}
```

```
dependencydefault == none;
dependencies {
    dataWriteOut.set_data() -> outDataAvailable;
}
```

```
dependencydefault == all;
dependencies {
    modeChange() ->;
    case modeChange.modeVar of {
        enabled: inDataAvailable -> dataIn.get_data(),
        outDataAvailable;
        disabled: inDataAvailable ->;
    }
}
```
**Light-weight Dependency Specs**

dependencydefault == all;
dependencies {
  modeChange() ->;
  case modeChange.modeVar of {
    enabled: inDataAvailable -> dataIn.get_data(),
    disabled: inDataAvailable ->;
  }
}  
behavior { ... }

**Component Behavior**

class BMModal {
  uses      ReadData dataIn;
  consumes  DataAvailable inDataAvailable;
  publishes DataAvailable outDataAvailable;
  provides  ReadData dataOut;
  provides  ChangeMode modeChange;
  enum Modes (enabled, disabled);
  Modes m;
  behavior {
    handles dataInReady (DataAvailable e) {
      case m of
        enabled  {
          dataOut::data <- dataIn.getData();
          push {} dataOutReady;
        } disabled {}  
    }
  }
}

**Outline**

4. Modal Behavior

Component Behavior

behavior for events on dataInReady port

**Behavior mode cases**

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL

behavior for events on dataInReady port

mode declaration using CORBA IDL
Component Behavior

```
class BMModal {
    uses ReadData dataIn;
    consumes DataAvailable inDataAvailable;
    publishes DataAvailable outDataAvailable;
    provides ReadData dataOut;
    provides ChangeMode modeChange;
    enum Modes (enabled, disabled);
    Modes m;
    behavior {
        handles dataInReady (DataAvailable e) {
            case m of
            | enabled  {
                dataOut::data <- dataIn.getData();
                push {} dataOutReady;
            } |
            | disabled {} |
        }
    }
}
```

Outline

```
4. Component Connections
```

Three Synchronized Views

```
Scenario Description
```

Textual View

```
Instance Airframe implements MlAirframe on Board2 { 
    connect this instance to the-Airframe.
    connect this data to the-GPS dataset.
}
```
Textual View

Graphical View

Design-level analyses model-base views

Spreadsheet View

Results of automatic rate group synthesis are fed back into spreadsheet

Popup menus give type-correct connection possibilities

Value added: Incremental, iterative scenario construction with multiple forms of visualization, analyses, and automated "design advice".
Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
  - component development
  - lightweight semantic annotations
  - intra-component dependences
  - intra-component transition semantics
- System assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor’s modeling language to support Cadena designs
- Customizing Bogor’s scheduling and state-space search modules to Cadena/BoldStroke designs

Leveraging Dependence Info

- Dependence declarations are leveraged in a variety of ways...
  - Form answers to visual queries about paths/dependences through configured systems
  - Provides info to automatic/smart placement of pieces of KSU Event Communication Framework middleware service
  - Basis of a number of forms of automated design advice (rate seeding, component distribution, etc.)

CCM Design Dependence Graphs

- From system configuration information
- From user-specified intra-component dependences
- State predicates giving conditional dependences

Aspect Synthesis

- Dependency-driven rate assignment to event handlers

- Automatic detection of optimization opportunities for asynchronous message delivery to synchronous method calls (must be co-located and run at same rate)
Optimizing Event Communication

Common Situation

if component is disabled, then event delivery of events from sensors is causing unnecessary overhead

Observe: if component is disabled, then all events flow through event channel, so...

Move mode logic into event channel

Event Channel

Optimizing Event Communication

Configurable Product Line Profiles

Cadena profiles enable flexible definition of attributes for CCM model entities and APIs for plug-in tools to access and manipulate attribute values.

CCM Development Support

Control: API for plugging CCM frameworks into Cadena for modeling, design, and analysis.

Data: Configuration data for D & C, Event Communication, etc.

CIAO Support

CIAO Support

CIAO build files

Auto-generate CIAO build files

...this info can be entered/synthesized at the modeling level

CIAO build files
**Lecture Outline**

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
  - component development
  - lightweight semantic annotations
  - intra-component dependences
  - intra-component transition semantics
- System assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor’s modeling language to support Cadena designs
- Customizing Bogor’s scheduling and state-space search modules to Cadena/BoldStroke designs

**Boeing Threading Model**

**Challenges of Event Communication**

Consider the following situation:

- Component C is receiving from two components A and B
- A and B send at different rates
- C needs both inputs to become active

We can:

- Reduce network traffic
- Simplify computation inside the component
- Clarify the design
- Define the components in a more general way
Middleware/Service Semantics

- Weak CCM and Event Services Specs (OMG)
  - Informal: English and examples
  - Intentionally under-specified to allow implementor freedom
  - Looked at implemented semantics of existing ORBs and Event Services
    - ACE/TAO, FACET, OpenCCM, ...
  - Developed a family of semantic models that captured their behavior
  - Implemented these models as Bogor extensions
    - model modules are reused each time we reason about a system

Domain-Specific Modeling

Bogor -- Extensible Modeling Language

```
Event publish() {
  Bogor API calls...
}
```

```
Event connect() {
  Bogor API calls...
}
```

Core Modeling Language
New Bogor primitives corresponding to Event Channel API
Java implementation of new primitives inside model-checker

Bogor Customized To Cadena

Bogor -- Extensible Modeling Language

```
Event publish() {
  Bogor API calls...
}
```

```
Event connect() {
  Bogor API calls...
}
```

Core Modeling Language
New Bogor primitives corresponding to Event Channel API
Java implementation of new primitives inside model-checker

Bogor Modeling Extensions

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```

```
extension Bogor for Cadena {Module}
  type Declarative {
    event_type : Bogor-event
  }
```

```
extension CCM for Cadena {Module}
  type Declarative {
    event_type : CCM-event
  }
```

```
extension ESB for Cadena {Module}
  type Declarative {
    event_type : ESB-event
  }
```

```
extension CORBA for Cadena {Module}
  type Declarative {
    event_type : CORBA-event
  }
```
**Domain-Specific Algorithms**

**Bogor -- Customizable Checking Engine Modules**

Bogor default modules are unplugged and replaced with state representation, scheduling and search strategies customized to the Bold Stroke domain.

**Assessments of Previous Work**

<table>
<thead>
<tr>
<th></th>
<th>Cadena</th>
<th>dSPI N (ICSE'02)</th>
<th>Bogor (FMCO'02)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing ModalSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 rate groups</td>
<td></td>
<td></td>
<td>1.4 M states</td>
</tr>
<tr>
<td>8 components</td>
<td></td>
<td></td>
<td>58 sec</td>
</tr>
<tr>
<td>125 events per hp</td>
<td></td>
<td></td>
<td>9.1 K states</td>
</tr>
<tr>
<td>Boeing MediumSP</td>
<td></td>
<td></td>
<td>8.59 sec</td>
</tr>
<tr>
<td>2 rate groups</td>
<td></td>
<td></td>
<td>21.5 MB</td>
</tr>
<tr>
<td>50 components</td>
<td></td>
<td></td>
<td>1.61 MB</td>
</tr>
<tr>
<td>820 events per hp</td>
<td></td>
<td></td>
<td>2 min</td>
</tr>
</tbody>
</table>

- want to check larger model
- does not seem to scale well regardless aggressive reductions

**Key Observation**

Leverage patterns of periodic computation
- use the structure of periodic systems to systematically drop states

**Leveraging Periodic Structure**

**Periodic Tasks**

<table>
<thead>
<tr>
<th>10 Hz</th>
<th>5 Hz</th>
<th>1 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hyper-period

"Macro-state" $S_1$

Bogor (FMCO'02)

**Leveraging Periodic Structure**

**Periodic Tasks**

<table>
<thead>
<tr>
<th>10 Hz</th>
<th>5 Hz</th>
<th>1 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hyper-period

"Macro-state" $S_1$

Basic Idea
- break the search into several regions
- divide the problem into smaller problems

**Quasi-Cyclic Structure**

Trace Structure

Macro-states

These successive macro-states maybe different (acyclic)...

...but a portion of each of the states is repeating...

...and so we say that the state-space is *quasi-cyclic*. 

**Macro-state Structure**

- Same at each macro-state:
  - dispatch queues empty, threads idle, correlators are at initial state
  - Different: component/system mode values are different
Many applications with control-loops have this property:
- GUIs, web-servers, ...

Use a predicate $\Phi$ to characterize the repeating portion.

Generalizing $\Phi$-conforming $\Phi$-states

Place initial $\Phi$-state in global store, and begin state exploration.

Place states in region state store until $\Phi$-state is encountered.

Place $\Phi$-state into global store.

Flush region state store.

Place states in region state store until $\Phi$-state is encountered.
Non-determinism in region generated two $\Phi$-states. Put these into global state store.

Flush region state store

Explore these regions until $\Phi$ states encountered

A Quasi-cyclic System: Example
Quasi-cyclic Search: Example

\( \Phi: pc = l_3 \land x = 0 \)

Global States = {}

Queues = {}

1: \( y = 0; \) goto l2;
2: \( x = 0; \) goto l3;
3: \( \text{true} \rightarrow x = 2; \) goto l4;
4: \( y + 5 \rightarrow \) skip; goto end;
5: \( y \leq 5 \rightarrow \) skip; goto l2;

end:

\( \Phi: pc = l_3 \land x = 0 \)

Global States = \{0\}

Queues = \{2\}

Quasi-cyclic Search: Example

\( \Phi: pc = l_3 \land x = 0 \)

Global States = \{0\}

Queues = \{3\}

Quasi-cyclic Search: Example

\( \Phi: pc = l_3 \land x = 0 \)

Global States = \{0,2\}

Queues = \{3,4\}

Quasi-cyclic Search: Example

\( \Phi: pc = l_3 \land x = 0 \)

Global States = \{0,2,3\}

Queues = \{4\}

Quasi-cyclic Search: Example

\( \Phi: pc = l_3 \land x = 0 \)

Global States = \{0,2,3,4\}

Queues = \{5\}
Quasi-cyclic Search: Example

Φ: pc = l3 ∧ x = 0

Global States = {0,2,3,4,5}
Queues = {}

l1: y = 0; goto l2;
l2: x = 0; goto l3;
l3: true -> x = 2; goto l4;
true -> x = 3; goto l4;
l4: y = y + x; goto l5;
l5: y > 5 -> skip; goto end;
y <= 5 -> skip; goto l2;
end:

Φ: pc = l3 ∧ x = 0

Global States = {0,2,3,4,5}
Queues = {}

Search each region independently
• max of 9 versus 37 states in classical DFS
• note that the sum here is >37
• same states may appear in multiple regions

Regions can be searched in parallel
• Works well when
• reasonable fraction of state variables are cyclic
• low-degree of overlapping between regions

Domain-Specific Algorithms

Verified Counter Example

Scaling Boeing ModalSP

both searches have exponential time growth
• quasi-cyclic search takes more time (overlapping regions)
• parallel quasi-cyclic takes 25% less time than classical DFS
• actively pursuing distributed solution
Conclusions

- Model-driven component-based development provides a variety of benefits
- One goal of system architecture design is to lift as much aspect logic up to modeling level as possible
  - System integrator (who assembles components together to form a system) plays a crucial "programming role" by selecting/configuring attributes and services
- Bogor can be customized to check Cadena system designs
  - customized scheduling and search strategies
  - new BIR extensions model the APIs of component infrastructure and RT-CORBA event channel

For More Information...

SAnToS Laboratory, Kansas State University
http://www.cis.ksu.edu/santos

Cadena Project
http://cadena.projects.cis.ksu.edu

...see us, for demo, examples, plug-in development, etc.