Software Model-checking:
The SAnToS/Bandera Perspective

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

Principal Investigators
Matt Dwyer
John Hatcliff

Postdocs and Students
Radu Iosif
Hongjun Zheng
Corina Pasareanu
Georg Jung

Robby
Venkatesh Ranganath
Oksana Tkachuk
William Deng

Support
US National Science Foundation (NSF)
US National Aeronautics and Space Agency (NASA)
US Department of Defense
Advanced Research Projects Agency (DARPA)
US Army Research Office (ARO)

Rockwell-Collins ATC
Honeywell Technology Center and NASA Langley
Sun Microsystems
Intel

The Bandera Perspective

This talk will focus on Bandera and Cadena and will give the Bandera/SAnToS perspective on software model-checking

For other perspectives see...

- Java PathFinder – JPF (NASA Ames)
- SLAM Project (Microsoft Research)
- BLAST Project (U. Berkeley)
- FeaVer Project (Lucent/Bell Labs)
- Alloy (MIT)
## Goals of the Project

I. **Provide platform for construction of and experimentation with technologies for model-checking concurrent Java software**

   - ... model-reduction techniques
     e.g., abstraction, slicing, compiler-based optimizations
   - ... model-checking engines
     e.g., explicit-state, symbolic
   - ... property specification languages
     e.g., temp logic, state machines

II. **Integration with commonly used design notations, methods, and processes**

   - ... UML artifacts, CCM
     e.g., checking, specification
   - ... automatic generation of synchronization code with dedicated checking
   - ... integration with development and certification of safety-critical systems.

III. **Evaluation using safety-critical military and civilian applications as well as non-critical popular open-source software**

## In This Talk...

- Challenges in model-checking software and how Bandera addresses these **(30 minutes)**
- Overview of Bandera tool architecture and functionality of primary components **(40 minutes)**
- --- break ---
- Specification Patterns **(20 minutes)**
- Modeling Avionics Software **(40 minutes)**
- Conclusions **(10 minutes)**
Goals

- Draw connections with earlier lectures and explain how various concepts and techniques are similar/different in software
- Highlight hard open problems related to software model-checking
- Share what I think are future trends in software model-checking and why we as a community have some reasons for being optimistic

Model Checking

Finite-state model

Temporal logic formula

\[ \square(\Phi \rightarrow \Omega) \]

Model Checker

OK or Error trace

Line 5: ...
Line 12: ...
Line 20: ...
Line 25: ...
Line 27: ...
Line 45: ...
Line 47: ...
What makes model-checking software difficult?

- Finite-state model
- Temporal logic formula
- OK
- Error trace

Problems using existing checkers:

- Model Construction
- Property specification
- State explosion
- Output interpretation

Model Construction Problem

Semantic gap:

- Programming Languages
  - methods, inheritance, dynamic creation, exceptions, etc.
- Model Description Languages
  - automata
Model Construction Problem

- Due to state explosion, model-checking should not be applied to an entire code base, but rather to a unit.
- In OO software, boundaries between units are usually messy:
  - references flow out of unit, and external components can change state of objects created in unit.
  - call-backs (in all GUI code).
  - tedious to identify interaction points and define stubs/drivers.

What makes model-checking software difficult?

- Problems using existing checkers:
  - Model Construction
  - Property specification
  - State explosion
  - Output interpretation
Property Specification Problem

Difficult to formalize a requirement in temporal logic

“Between the window open and the window close, button X can be pushed at most twice.”

...is rendered in LTL as...

\[
[[([\text{open} V \langle<\text{close}\rangle) ->
  ([\text{!pushX} V \langle!\text{close}\rangle) U
  (\text{close} V ([\text{pushX} V \langle!\text{close}\rangle) U
  (\text{close} V ([\text{pushX} V \langle!\text{close}\rangle) U
  (\text{close} V ([\text{!pushX} U \text{close}])))])])]]
\]

Property Specification Problem

Forced to state property in terms of model rather than source

We want to write source level specifications...

\[\text{Heap. b. head} = \text{Heap. b. tail}\]

We are forced to write model level specifications...

\[
(((\text{collect}(\text{heap_b}) = 1) \\
  \&\& (\text{BoundedBuffer}\_\text{col}\.\text{instance}[_\text{index}(\text{heap_b})].\text{head} = \\
  \text{BoundedBuffer}\_\text{col}\.\text{instance}[_\text{index}(\text{heap_b})].\text{tail}) ) \\
\| ((\text{collect}(\text{heap_b}) = 3) \\
  \&\& (\text{BoundedBuffer}\_\text{col}\_0\.\text{instance}[_\text{index}(\text{heap_b})].\text{head} = \\
  \text{BoundedBuffer}\_\text{col}\_0\.\text{instance}[_\text{index}(\text{heap_b})].\text{tail}) ) \\
\| ((\text{collect}(\text{heap_b}) = 0) \&\& \text{TRAP})
\]
**Property Specification Problem**

Complications arise due to the dynamic nature of OO software

Consider multiple instances of a bounded buffer class...

**Requirement:** If a buffer instance becomes full, it will eventually become non-full.

In general, a heap object has no program-level name that persists throughout the lifetime of the object.

---

**What makes model-checking software difficult?**

Finite-state model

Temporal logic formula

Model Checker

OK

Error trace

Problems using existing checkers:

- Model Construction
- Property specification
- State explosion
- Output interpretation
State Explosion Problem

Moore's law and algorithm advances can help
- Explosive state growth in software limits scalability

What makes model-checking software difficult?

- Model Construction
- Property specification
- State explosion
- Output interpretation
Output Interpretation Problem

Program

Model Description

Error trace

Raw error trace may be 1000's of steps long

Must map line listing onto model description

Mapping to source is made difficult by

- Semantic gap & clever encodings of complex features
- Multiple optimizations and transformations
- Over-approximations in abstractions may yield infeasible error traces (how to decide if feasible or not?)

Bandera:
An open tool set for model-checking Java source code

Graphical User Interface

Optimization Control

Checker

Inputs

Checker

Outputs

Transformation & Abstraction Tools

Error Trace Mapping

Bandera

Java Source

Bandera Temporal Specification
Addressing the Model Construction Problem

Model extraction: compiling to model checker inputs:

- Numerous analyses, optimizations, two intermediate languages, multiple back-ends
- Slicing, abstract interpretation, specialization
- Variety of usage modes: simple...highly tuned

Bandera Environment Generation Tools

- Identify classes in unit
- Automatically finds points of interaction (where unit calls outside classes or is called itself)
Addressing the Model Construction Problem

Bandera Environment Generation Tools
- Identify classes in unit
- Automatically finds points of interaction (where unit calls outside classes or is called itself)
- Cuts away non-unit classes
- Automatically generates driver (generates calls to unit based on regular expression or LTL formula)
- Automatically generates stubs

Addressing the Property Specification Problem

An extensible language based on field-tested temporal property specification patterns

```
[(open | <=close) ->
 (,!pushX | !close) U
 (close | (!pushX | !close) U
 (close | (!pushX | !close) U
 (close | (!pushX | !close) U
 (close | (!pushX U close)))]
```

Using the pattern system: **2-bounded existence**

Between \{open\} and \{close\}
\{pushX\} exists atMost \{2\} times;
**Addressing the State Explosion Problem**

- Result: multiple models
  - even as many as one per property
- Aggressive customization via slicing, abstract interpretation, program specialization

**Generate models customized wrt property!**

**Addressing the Output Interpretation Problem**

- Run error traces forwards and backwards
- Program state queried
- Heap structures navigated & visualized
- Locks, wait sets, blocked sets displayed

Like a debugger: error traces mapped back to source
class BoundedBuffer {
    Object [] buffer;
    int head;  /* next available slot */
    int tail;  /* last available slot */
    int bound; /* max # of elements */

    public BoundedBuffer(int b)
    {..}

    public synchronized boolean isEmpty()
    {..}

    public synchronized void add(Object o)
    {..}

    public synchronized Object take()
    {..}
}
Property Specification

/**
* @observable
* EXP Full: \( \text{head} = \text{tail} \);
*/
class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public synchronized
    void add(Object o)
    {
    }

    public synchronized
    Object take()
    {
    }
}

Requirement:
If a buffer becomes \text{full}, it will eventually become \text{non-full}.

Bandera Specification:
FullToNonFull:
forall\[b:\text{BoundedBuffer}\].
\{\text{Full}(b)\} leads to \{\neg\text{Full}(b)\} globally;

Property Specification

/**
* @observable
* EXP Empty:
* \( \text{head} = ((\text{tail} + 1) \mod \text{bound}) \);
*/
class BoundedBuffer {
    int head, tail, bound;
/**
* @observable INVOKE Call;
*/
    public synchronized
    void add(Object o)
    {
    }

/**
* @observable RETURN Return;
*/
    public synchronized
    Object take()
    {
    }
}

Requirement:
\text{Empty buffers} must added to before being taken from

Bandera Specification:
NoTakeWhileEmpty:
forall\[b:\text{BoundedBuffer}\].
\{\text{take. Return}(b)\} is absent after \{\text{Empty}(b)\}
until \{\text{add. Call}(b)\};
Quantification

forall[b:BoundedBuffer].P(b)

- Quantified set BoundedBuffer is not fixed
  - varies within executions
  - varies across executions

- Solution
  - add a state variable (for b) that will eventually be bound non-deterministically to each instance
  - by enabling checking of the formula only when variable is bound to an instance

Quantification (Cont'd)

(!\text{selected} \lor (\text{selected} \land P(b))) \lor []!\text{selected}

Original Model

1. new BoundedBuffer (n)
2. new BoundedBuffer (n)
3. new BoundedBuffer (k)

Augmented Model

1. selected
2. selected
3. selected

1. selected
2. selected
3. selected

new BoundedBuffer (n)
new BoundedBuffer (n)
new BoundedBuffer (k)
new BoundedBuffer (k)
Quantification (Cont'd)

Original Model

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public BoundedBuffer(int n) {
        ...
    }
}

Augmented Model

class heap {
    public static BoundedBuffer b;
}

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public BoundedBuffer(int n) {
        if (heap.b == null &&
            Bandera.choose()) {
            heap.b = this;
        } else {
        }
    }

Quantification (Cont'd)

forall[b:BoundedBuffer].
    {Full(b)} leads to {!Full(b)} globally;

Bandera compiles to...

( heap.b == null ) U
( heap.b != null &&
  ((*)(heap.b.head == heap.b.tail) ->
   (heap.b.head != heap.b.tail)))
|| ([](heap.b == null)
Front End

```java
public synchronized void add(Object o) {
    while (tail == head)
        try { wait(); } catch (InterruptedException ex) {};
    buffer[head] = o;
    head = (head + 1) % bound;
    notifyAll();
}
```

Java

```
public synchronized void add(java.lang.Object) {
    T$0 := @this;
    o := @parameter0;
    entermonitor T$0;
    label0: goto label4;
    label1: virtualinvoke T$0.[wait():void]();
    T$3 = T$0.[head:int];
    T$4 = T$0.[buffer: Object[]];
    T$4[T$3] = o;
    T$0.[notifyAll():void]();
}
```

Jimple (excerpts)

Property-directed Slicing

- slicing criterion generated automatically from observables mentioned in the property
- backwards slicing automatically finds all components that might influence the observables
Property-directed Slicing

```java
/**
 * @observable EXP Full: (head == tail)
 */

class BoundedBuffer {
    Object[] buffer_;
    int bound;
    int head, tail;

    public synchronized void add(Object o) {
        while (tail == head)
            try { wait(); } catch (InterruptedException ex) {};
        buffer_[head] = o;
        head = (head+1) % bound;
        notifyAll();
    }
...
}
```

Slicing Criterion
All statements that assign to head, tail.

Abstraction Engine
Collapses data domains via abstract interpretation:

Code
```java
int x = 0;
if (x == 0)
    x = x + 1;
```

Data domains
```
int
(n<0) : neg
(n==0) : zero
(n>0) : pos
```
Abstraction Component Functionality

<table>
<thead>
<tr>
<th>Variable</th>
<th>Concrete Type</th>
<th>Abstract Type</th>
<th>Inferred Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int</td>
<td>Signs</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>Signs</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>bool</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>int</td>
<td>intAbs</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>Object</td>
<td>Point</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Buffer</td>
<td>Buffer</td>
<td></td>
</tr>
</tbody>
</table>

Abstraction Library

Abstraction Engine

Abstracted Jimple

PVS

BASL Compiler

Bandera Abstraction Specification Language

Jimple

Example: Start safe, then refine: 
\[ +(\text{NEG},\text{NEG}) = \{\text{NEG}, \text{ZERO}, \text{POS} \} \]

Proof obligations submitted to PVS...

For all \( n_1, n_2 \): \( \text{neg?(} n_1 \text{)} \) and \( \text{neg?(} n_2 \text{)} \) implies not \( \text{pos?(} n_1 + n_2 \text{)} \) ✓

For all \( n_1, n_2 \): \( \text{neg?(} n_1 \text{)} \) and \( \text{neg?(} n_2 \text{)} \) implies not \( \text{zero?(} n_1 + n_2 \text{)} \) ✓

For all \( n_1, n_2 \): \( \text{neg?(} n_1 \text{)} \) and \( \text{neg?(} n_2 \text{)} \) implies not \( \text{neg?(} n_1 + n_2 \text{)} \) ❌
Compiling In Abstractions

```
abstraction Sign abstracts int
TOKENS = { NEG, ZERO, POS };
abstract(n)
begin
  n < 0   -> {NEG};
  n == 0  -> {ZERO};
  n > 0   -> {POS};
end
operator + add
begin
  (NEG, NEG)  -> {NEG}  ;
  (NEG, ZERO) -> {NEG}  ;
  (ZERO, NEG)  -> {NEG}  ;
  (ZERO, ZERO) -> {ZERO} ;
  (ZERO, POS)  -> {POS}  ;
  (POS, ZERO)  -> {POS}  ;
  (POS, POS)  -> {POS}  ;
  (_,_)        -> {NEG, ZERO, POS};
/* case (POS,NEG),
   (NEG,POS) */
end
```

```
public class Signs {
  public static final int NEG  = 0; // mask 1
  public static final int ZERO = 1; // mask 2
  public static final int POS  = 2; // mask 4

  public static int abstract(int n) {
    if (n < 0) return NEG;
    if (n == 0) return ZERO;
    if (n > 0) return POS;
  }

  public static int add(int arg1, int arg2) {
    if (arg1==NEG  && arg2==NEG)  return NEG;
    if (arg1==NEG  && arg2==ZERO) return NEG;
    if (arg1==ZERO && arg2==NEG)  return NEG;
    if (arg1==ZERO && arg2==ZERO) return ZERO;
    if (arg1==ZERO && arg2==POS)  return POS;
    if (arg1==POS  && arg2==ZERO) return POS;
    if (arg1==POS  && arg2==POS)  return POS;
    return Bandera.choose(0,2);
    /* case (POS,NEG), (NEG,POS) */
  }
}
```
Comparing Traces

Choice-bounded Search

Detectable Violation
Undetectable Violation
State space searched

choose()
Property Abstraction

Goal:
If the abstract property holds on the abstract system, then the original property holds on the original system.

Basic Idea
- Property (LTL) is converted to negation-normal form.
- For each predicate (e.g., on integers) of the form \( P(x, c) \) where \( x \) is bound to abstraction \( A \), we replace \( P(x, c) \) by a disjunction of cases that guarantee \( P(x, c) \) to be true.

Examples (where \( x \) is bound to Signs)

\[
\begin{align*}
[](x > 0) & \quad \text{abstracted to } [](x == \text{pos}) \\
[](x > -2) & \quad \text{abstracted to } [](x == \text{zero} || x == \text{pos})
\end{align*}
\]
These two states should be considered equal, but they have different representations.
Heap Issues

Different thread interleavings may cause different positioning of heap objects. This will cause observationally equivalent heaps to be considered distinct states -- leading to tremendous state explosion.

For avoiding state-space explosion when model-checking OO software, one needs a heap representation that identifies as many observationally equivalent heaps as possible!

Simple Representation

For avoiding state-space explosion when model-checking OO software, one needs a heap representation that identifies as many observationally equivalent heaps as possible!
Bounded Buffer BIR

State Declarations

process BoundedB() -- static identification of threads
  BoundedBuffer_rec = record {
    bound : range -1..4;  -- bounded integer values
    head  : range -1..4;  -- qualified lock representation
    tail  : range -1..4;
    BIRLock : lock wait reentrant;
  };


BoundedBuffer_ref = ref { BoundedBuffer_col, BoundedBuffer_col_0 };  -- one per allocator site

Reference type indicates mini-heaps that can be pointed to.
Easily express results of “points-to” analysis

BIR Transitions

loc s34:  -- control point label
  live { b2, b1, T_0, T_6, T_8 }  -- live variable information
  when true
  do invisible {
    T_8 := (T_6 % T_8);
  } goto s35;
...

loc s36:
  live { b2, b1, T_0 }
  when true do {
    notifyAll(T_0.BIRLock);
  } goto s37;
...

loc s37:
  live { b2, b1, T_0 }
  when true do {
    unlock(T_0.BIRLock);
  } goto s38;  -- built-in operations on lock representations
Bounded Search Strategies

Usual strategy
- Carry out depth/breadth-first search to depth $k$

Bandera strategy
- Carry out search until resources from particular classes are exhausted
- integer size, # instances at each allocator site, # processes, # activation frames
Bounded Buffer Promela

```c
typedef BoundedBuffer_rec
{ type_8 bound;
  type_8 head;
  type_8 tail;
  type_18 BIRLock; }

... ...
loc_25:
atomic {
  printf("BIR: 25 0 1 OK \n");
  if :: (_collect(T_0) == 1) ->
    T_B = BoundedBuffer_col.
    instance[_index(T_0)].tail;
  :: (_collect(T_0) == 2) ->
    T_B = BoundedBuffer_col_0.
    instance[_index(T_0)].tail;
  :: else ->
    printf("BIR: 25 0 1 NullPointerException \n");
    assert(0);
  fi;
  goto loc_26;
}
```

... record implementation

... BIR AST markers get printed with error trail. Parsed and drive BIR simulator for counter-example display.

... Accessing mini-heaps for buffer tail component.

... Bounded Buffer Promela

... dSpin Backend

Different thread interleavings may cause different positioning of heap objects. This will cause observationally equivalent heaps to be considered distinct states - leading to tremendous state explosion.

garbage collection & canonical ordering on objects based on lexicographical order on field names in reachability chain

Observationally Equivalent

Canonical Heap
(fully abstract)
Case Study

Honeywell Digital Engine Operating System (DEOS)
- A real-time operating system for integrated modular avionics systems
- Demonstration artifact for NASA Langley funded project on incorporating formal methods in FAA certification
- DEOS Scheduler: non-trivial concurrent Java program: 1443 lines of code, 20 classes, 6 threads

Verification of Abstracted DEOS

Time Partitioning Requirement:
Application processes are guaranteed to be scheduled for their budgeted time during a scheduling unit (known bug)

- Bandera Abstraction & JPF
  - Bandera’s dependence graph used to identify relevant controlling conditional expressions
  - produced a 464 step counter-example
- Using non-determinism bounded search
  - found a guaranteed feasible 318 step counter-example
- After fixing the bug, the requirement was verified
  - ~15 min
Summary

Bandera is an open platform for experimentation

- Designed for extensibility
  - Well-defined internal representations and interfaces
  - We hope this will contribute to the definition of APIs for software model-checkers and associated tools
- Tutorial, example repository, lecture slides, etc. on web-site
- Current release is usable on relatively small examples, but not robust enough for industrial use or large semester-long projects.
- Updated, more robust implementation in mid-September and mid-November
- Complete rewrite of tool to obtain robust implementation with very good user-interface coming early 2003.

Challenging Open Problems

- Compositional model-checking for concurrent OO systems
  - Issues with references, dynamic data make the OO setting light-years beyond settings used in current foundational work
  - If we scale down the properties we want to check (e.g., to interface protocols) then there is more hope.
Challenging Open Problems

- Automated abstraction and refinement techniques in the presence of dynamically allocated data and concurrency
  - SLAM and BLAST have shown how automated abstraction and refinement can be effective for sequential code with primarily integer manipulation.
  - Work on three-valued logic (TVLA) provides a nice foundation for heap-abstraction, but automated counter-example driven refinement is still a challenge.

Strategies for Moving Forward

- Trojan-horse formal methods, e.g.
  - FDR/Refinement checking in UML RT
  - SLAM in device-driver certification tool-kit
  - Software model-checkers integrated with robust testing and debugging infrastructures

- Combine model-checking of design artifacts (these provide system abstractions) with refinement checking of code against designs
  - In large systems, getting the overall design correct is more difficult/important than crunching out the implementation of your classes
  - Tools like a scaled-up Alloy attached to UML or other design artifacts could be very useful