Bandera: Extracting Finite-state Models from Java Source Code

James C. Corbet  Matthew B. Dwyer  John Hatcliff
Shawn Laubach  Corina S. Păsăreanu  Robby  Hongjun Zheng

Presenter: Henrik Kragh-Hansen

October 12, 2007
Motivation

Implementation

Finite-state Model

Verification
Motivation

Implementation

Bandera

Finite-state Model

Verification
## Existing verification tools

### Finite-state verification

- Ensure properties
- Verifies the design
- Cost-effective
How finite-state models are constructed

Manually constructed models

- Expensive
- Prone to errors
- Difficult to optimize
## Problems

### Model construction problem
- Developed in general-purpose languages
- Must extract an abstract model
- Specific input language of the tool

### State explosion problem
- Exponential increase in the size of a finite-state model
- Software tends to have more states than hardware components
Solution

Bandera

- Java source code as input
- Generates a finite-state model
- Supports existing verification tools
- Maps result to original source code
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandera Design</td>
</tr>
<tr>
<td>Bandera Components</td>
</tr>
<tr>
<td>Bandera in Practice</td>
</tr>
<tr>
<td>Conclusion</td>
</tr>
</tbody>
</table>
### Design Criteria

<table>
<thead>
<tr>
<th>Reuse of existing checking technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Several existing products</td>
</tr>
<tr>
<td>• Existing products are highly optimized</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Automated support for the abstraction used by experienced model designers</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Not just be a translation</td>
</tr>
<tr>
<td>• Optimize the generated model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specialized models for specific properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Customized models for a particular property</td>
</tr>
</tbody>
</table>
Design Criteria

An open design for extensibility
- Loosely connected components
- Intermediate representations
- Allow new abstraction techniques and checking engines

Synergistic integration with existing testing and debugging techniques
- Counterexamples not specific to the model checker
Model Generation

Irrelevant component elimination
- Many components may not be relevant for the property being verified

Data abstraction
- Range of variables may be abstracted

Component restriction
- Constructing a restricted model
- Bound the amount of objects created
- Bound the number of total execution steps
Bandera

Components of Bandera

- Slicer
- Abstraction Engine
- Back End
- User Interface
Bandera’s Interface and Internals

**Abstraction Spec**
class Signs extends Int {
    Tokens = {Zero, Pos, Neg};
    Pos+Pos -> Pos;
}

**Restriction**
max Stage is 3;

**Abstraction Binding**
abstract Connector.queue with Signs;

**Specification**
[](Heap.c1.queue>0 ->
<>Stage1.run:return)

**Java Source**
public class Stage1 extends Thread {
    public void run() {...}
}

**Counterexample**
Main.main#3: (new Stage1()).

Bandera’s Interface and Internals

- **Bandera’s Interface and Internals**
- **Intermediate**
- **Constructor**
- **SMV**
- **SPIN**
- **SAL**
- **Abstraction**
- **Library**
- **Supplementary Analyses**
- **Proof Obligations**
- **Back End**
- **BASL**
- **BIR-Jimple-Java**
- **Tracer**
- **Slicer**
- **Abstraction-Based Specializer**
- **Property Front-end**
- **Java Front-end**
- **Java**
- **Primitive Propositions**
- **Abstraction Specifier**
- **Abstraction Binding**
- **Specification**
- **Java Source**
- **Counterexample**

PVS -> SPIN

SPIN Trans
SMV Trans
SAL Trans

BIR, BIRC, BIR-Jimple-Java, Tracer, Jimple, Slicer, Abstraction-Based Specializer, Property Front-end, Java Front-end, Java, Primitive Propositions, Abstraction Specifier, Abstraction Binding, Specification, Java Source, Counterexample

12/28
Soot/Jimple

**Soot framework**
- Translates Java programs to Jimple
- Java-to-Jimple-to-Java Compiler (JJJC)
- Bandera Intermediate Representation (BIR)
- Java $\leftrightarrow$ Jimple $\leftrightarrow$ BIR
The Slicer

Slicer

- Statements of interest $C = s_1, ..., s_k$ in program $P$
- Slicer removes statements from $P$ not affecting $C$
- Checking against specification $\phi$
- $\phi$ holds for $P$ iff $\phi$ holds for the reduced version of $P$
- The reduction is sound and complete
Example

$$[(\text{Heap.c1.queue} > 0 \rightarrow \text{<>Stage1.run: return})]$$

The slicing algorithm guarantees

- Preserves program components affecting assignments to \text{Heap.c1.queue}
- The relative order of execution of \text{Stage1.run}'s return and assignments to \text{Heap.c1.queue}
Abstraction Engine

Abstraction-Based Specializer

- Reduces model size via data abstraction
- Checking whether item is in vector is abstracted to a small set
  - \{ItemInVector, ItemNotInVector\}
- Some information is lost
- The user can choose abstraction for relevant variables
# Back End

- Sliced and abstracted program as input
- Verifier-specific representations as output
- Through Bandera Intermediate Representation (BIR)
- Extensible to new verifiers
- Contains high-level constructs
## Supplementary Analyses

- Improves the effectiveness of Bandera components by static analysis
- Results in more accurate and compact models
- E.g. a lock safety analysis
Bandera’s User Interface

<table>
<thead>
<tr>
<th>Formalism:</th>
<th>Pattern:</th>
<th>Scope:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL</td>
<td>Absence</td>
<td>Globally</td>
</tr>
</tbody>
</table>

- **Source Window**
- **Property Specification Window**
- **Counter Example Window**
Manual Abstraction

Variable abstractions

- Easy to select variables for abstraction
- Optimal abstractions are difficult
- Naive abstractions often effective
class Heap{
    static Connector c1,c2,c3,c4;
}
class Main{
    static public void main(
        String argv[]){
    Heap.c1 = new Connector();
    Heap.c2 = new Connector();
        (new Stage1()).start();
    Heap.c3 = new Connector();
        (new Stage2()).start();
    Heap.c4 = new Connector();
        (new Stage3()).start();
        (new Listener()).start();
    for (int i=1; i<10; i++)
        Heap.c1.add(i);
    Heap.c1.stop();
    }
}

class Connector{
    public synchronized int take(){ .. }
    public synchronized void add(int o){ .. }
    public synchronized void stop(){ .. }
}
class Stage1 extends Thread{
    public void run(){
        int tmp = -1;
            while (tmp != 0)
                if ((tmp=Heap.c1.take()) != 0)
                    Heap.c2.add(tmp+1);
                    Heap.c2.stop();
    }
}
class Stage2 extends Thread { .. }
class Stage3 extends Thread { .. }
class Listener extends Thread { .. }
class Heap{
    static Connector c1,c2,c3,c4;
}
class Main{
    static public void main(
        String argv[]){
            Heap.c1 = new Connector();
            Heap.c2 = new Connector();
            (new Stage1()).start();
            Heap.c3 = new Connector();
            (new Stage2()).start();
            Heap.c4 = new Connector();
            (new Stage3()).start();
            (new Listener()).start();
            for (int i=1; i<10; i++)
                Heap.c1.add(i);
            Heap.c1.stop();
        }
    }

class Connector{
    public synchronized int take(){ .. }
    public synchronized void add(int o){ .. }
    public synchronized void stop(){ .. }
}
class Stage1 extends Thread{
    public void run(){
        int tmp = -1;
        while (tmp != 0)
            if ((tmp = Heap.c1.take()) != 0)
                Heap.c2.add(tmp+1);
        Heap.c2.stop();
    }
}
class Stage2 extends Thread { .. }
class Stage3 extends Thread { .. }
class Listener extends Thread { .. }
## Bandera Pipeline Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Extract Time (s)</th>
<th>Check Time (s)</th>
<th>Check Result</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>b,r1,n</td>
<td>24</td>
<td>2674</td>
<td>true</td>
<td>7338120</td>
</tr>
<tr>
<td>b,r1,s</td>
<td>13</td>
<td>4</td>
<td>true</td>
<td>3748</td>
</tr>
<tr>
<td>b,r1,a</td>
<td>15</td>
<td>4</td>
<td>true</td>
<td>895</td>
</tr>
<tr>
<td>b,r2,s</td>
<td>13</td>
<td>56</td>
<td>true</td>
<td>528059</td>
</tr>
<tr>
<td>b,r2,a</td>
<td>16</td>
<td>11</td>
<td>true</td>
<td>27519</td>
</tr>
<tr>
<td>b,p1,s</td>
<td>13</td>
<td>4</td>
<td>true</td>
<td>2507</td>
</tr>
<tr>
<td>b,p1,a</td>
<td>15</td>
<td>4</td>
<td>true</td>
<td>331</td>
</tr>
<tr>
<td>d,r1,s</td>
<td>13</td>
<td>3</td>
<td>false</td>
<td>88</td>
</tr>
<tr>
<td>d,r1,a</td>
<td>15</td>
<td>2</td>
<td>false</td>
<td>17</td>
</tr>
</tbody>
</table>
Results

What does this data illustrate?

- Improves scalability of verification tools
- Extraction of compact models is crucial
- Counter examples must be reduced
  - E.g. reduced from 1780 to 159 steps
Related Work

Related work
- JavaPathFinder
- JCAT
- Feaver
- Erlang model-checker
The Main Contributions of the Paper

<table>
<thead>
<tr>
<th>The description of Bandera</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Slicer</td>
</tr>
<tr>
<td>- Abstraction Engine</td>
</tr>
<tr>
<td>- Back End</td>
</tr>
<tr>
<td>- User Interface</td>
</tr>
</tbody>
</table>
# My Opinion

## Pros
- Automatic generation of Finite-state model
- Currently supports SPIN as verifier
- Extensible with new verifiers
- Component based
- Cited by 127

## Cons
- Lack details
- Does only support a subset of Java
- Real-time?
What now?

Development since the article was published

- Other tools
  - Cadana
  - Bogor
- Current version 1.0 only as CLI
  - Trying to use Bogor as model checker
  - Eclipse plug-in
- Currently not being developed/funded
Thank you for your Attention

Any questions?