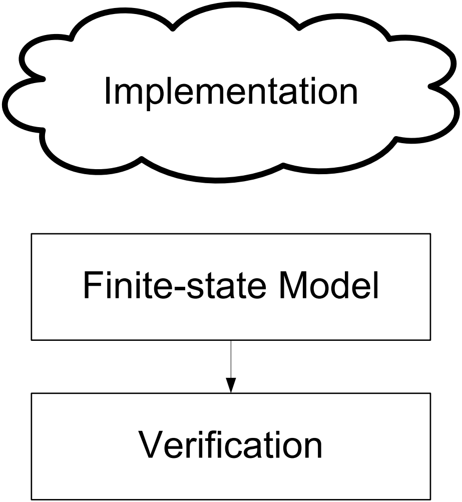
Bandera: Extracting Finite-state Models from Java Source Code

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

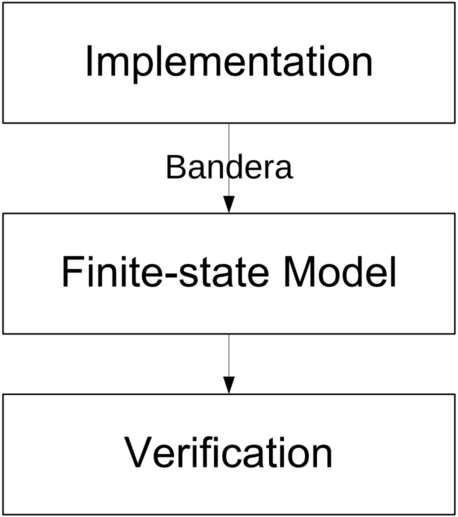
Presenter: Henrik Kragh-Hansen

October 12, 2007

Motivation



Motivation



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

Outline

1. Bandera Design
2. Bandera Components
3. Bandera in Practice
4. Conclusion

Design Criteria

Reuse of existing checking technologies

- Several existing products
- Existing products are highly optimized

Automated support for the abstraction used by experienced model designers

- Not just be a translation
- Optimize the generated model

Specialized models for specific properties

- Customized models for a particular property

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

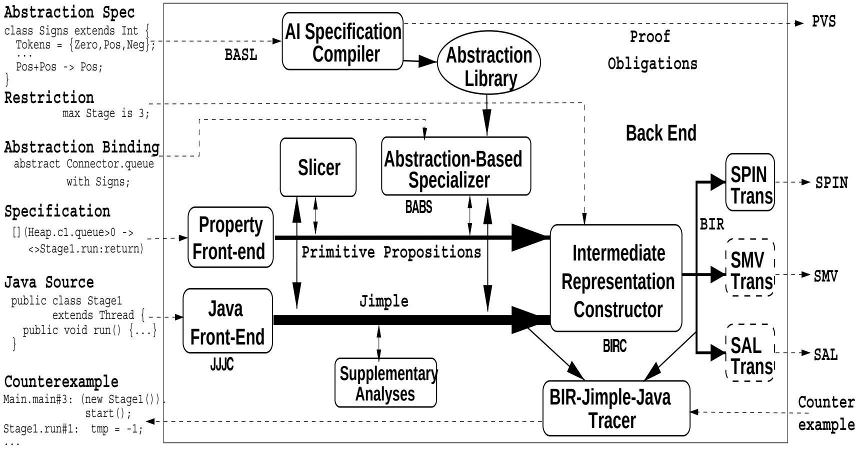
- Constructing a restricted model
- Bound the amount of objects created
- Bound the number of total execution steps

Bandera

Components of Bandera

- Slicer
- Abstraction Engine
- Back End
- User Interface

Bandera's Interface and Internals



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 ϕ
- ϕ holds for P iff ϕ holds for the reduced version of P
- The reduction is sound and complete

Example

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

The slicing algorithm guarantees

- Preserves program components affecting assignments to `Heap.c1.queue`
- The relative order of execution of `Stage1.run`'s return and assignments to `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

Back End

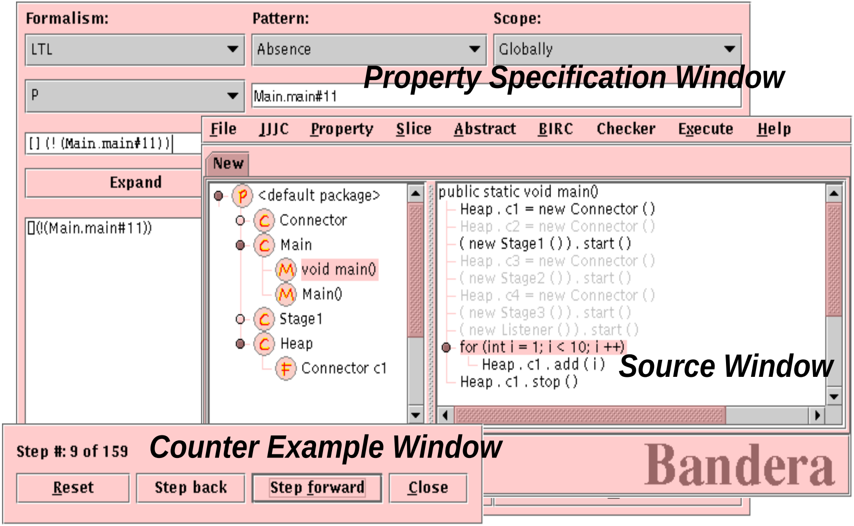
- Sliced and abstracted program as input
- Verifier-specific representations as output
- Through Bandera Intermediate Representation (BIR)
- Extensible to new verifiers
- Contains high-level constructs

Supplementary Analyses

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



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

Problem	Extract Time (s)	Check Time (s)	Check Result	States
b,r1,n	24	2674	true	7338120
b,r1,s	13	4	true	3748
b,r1,a	15	4	true	895
b,r2,s	13	56	true	528059
b,r2,a	16	11	true	27519
b,p1,s	13	4	true	2507
b,p1,a	15	4	true	331
d,r1,s	13	3	false	88
d,r1,a	15	2	false	17

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

The description of Bandera

- Slicer
- Abstraction Engine
- Back End
- User Interface

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?