Software Testing and Validation
Presentation

FINDING FEASIBLE COUNTER-EXAMPLES WHEN MODEL CHECKING ABSTRACTED JAVA PROGRAMS

Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser

Group 11 Members:
José Maria Costa Luís Cardoso nº99096
Luís Filipe Pedro Marques nº99265
Yassir Mahomed Yassin nº100611

INTRODUCTION

- **Challenges in applying model checking to software**

    State explosion problem

- **Need for abstraction and its consequences**

    Property-preserving abstraction

    Spurious behaviors and counter-example analysis

Objective: Automate the analysis of counter-examples from abstract model checks to identify real defects

BACKGROUND

- **Bandera toolset**

Program slicing and data abstraction

Integration with model checkers (Spin, SMV, JPF)

- **Java PathFinder (JPF) model checker**

Custom-built Java Virtual Machine (JVM)

Optimized for reducing state space

Granularity of atomic steps (byte-codes, source lines, explicit atomic blocks)

Counter-example generation and simulation

Goal: Leverage Bandera and JPF to automate counter-example analysis in abstracted Java programs

PROGRAM ABSTRACTION PROCESS

Four main steps:

1.Define an abstraction mapping
2. Transform the concrete program and property
3. Verify the abstract program
4. Infer the result for the concrete program

- **Data abstraction using Abstract Interpretation**

  Abstract domain, abstraction function, and abstract operations

- **Property abstraction**

  Converting properties to disjunctions of abstract propositions

  Under-approximation to ensure soundness

- **Scheduler abstraction**

  Most general scheduling policy: nondeterministic choice among runnable threads

  Preserving properties under more restrictive policies

EXAMPLE: SIGN ABSTRACTION

- **Signs abstraction:** tracks whether an integer is negative, zero, or positive
- **Abstract domain:** {neg, zero, pos}
- **Abstraction function:** maps concrete values to abstract tokens
- **Abstract operations:** respect the abstract domain (e.g., addition)
- **Bandera implementation:** replaces concrete Java operations with calls to abstract class methods

$+_{abs}$	$zero$	pos	neg
$zero$	$zero$	pos	neg
pos	pos	pos	$\{zero, pos, neg\}$
neg	neg	$\{zero, pos, neg\}$	neg

SOLUTION OVERVIEW

Integration of two tools: Bandera and Java Pathfinder

Application of two techniques: choose-free state space search and abstract counterexample guided concrete simulation

CHOOSE-FREE STATE SPACE SEARCH

Goal: Ensure that the counterexamples we find during model checking are actually feasible in the concrete program, and not just artefacts of the abstraction.

JPF will seach the program's state space but only along paths that do not involve nondeterministic choices.

By restricting the search we can ensure that any counterexample is relevant and feasible.

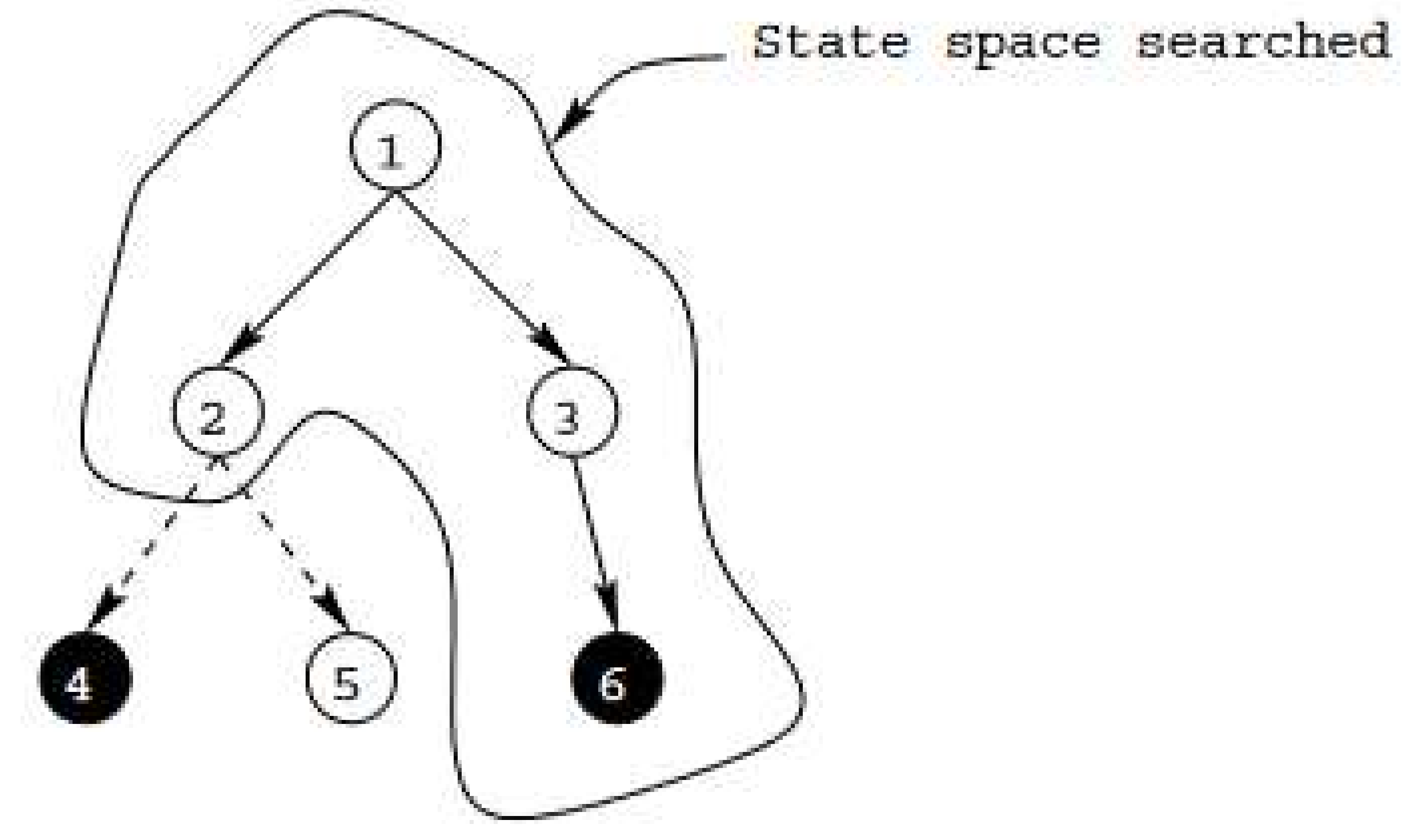

Fig. 2. Model Checking on Choose-free Paths

Alterations to JBF to follow the **Theorem:**

Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.

CHOOSE-FREE STATE SPACE SEARCH

Steps:

- Bandera creates an abstract model
- JPF tries and find counter examples
- JPF performs Choose-free state space search
- Confirmation of feasible counterexamples if a property violation is found

```
    class App{
     public static void main(...){
[1]   new AThread().start(); ...
[2]   int i=0;
[3]   while(i<2){...
[4]     assert(!Global.done);
[5]     i++;
     }}}
    class AThread extends Thread{
     public void run(){ ...
[6]   Global.done=true;
     }}
```

```
class App{
 public static void main(...){
  new AThread().start(); ...
  int i=Signs.ZERO;
  while(Signs.lt(i,Signs.POS)){...
   assert(!Global.done);
   i=Signs.add(i,Signs.POS);
  }}}
class AThread extends Thread{
 public void run(){ ...
  Global.done=true;
 }}
```

Fig. 3. Simple Example of Concrete (left) and Abstracted (right) Code

ABSTRACT COUNTER-EXAMPLE GUIDED CONCRETE SIMULATION

Steps:

- Model checking to identify an abstract counterexample
- JPF guides the simulation of the concrete program using the counterexample
- Tries to find a corresponding trace in the concrete execution that matches the abstract counterexample

ABSTRACT COUNTER-EXAMPLE GUIDED CONCRETE SIMULATION

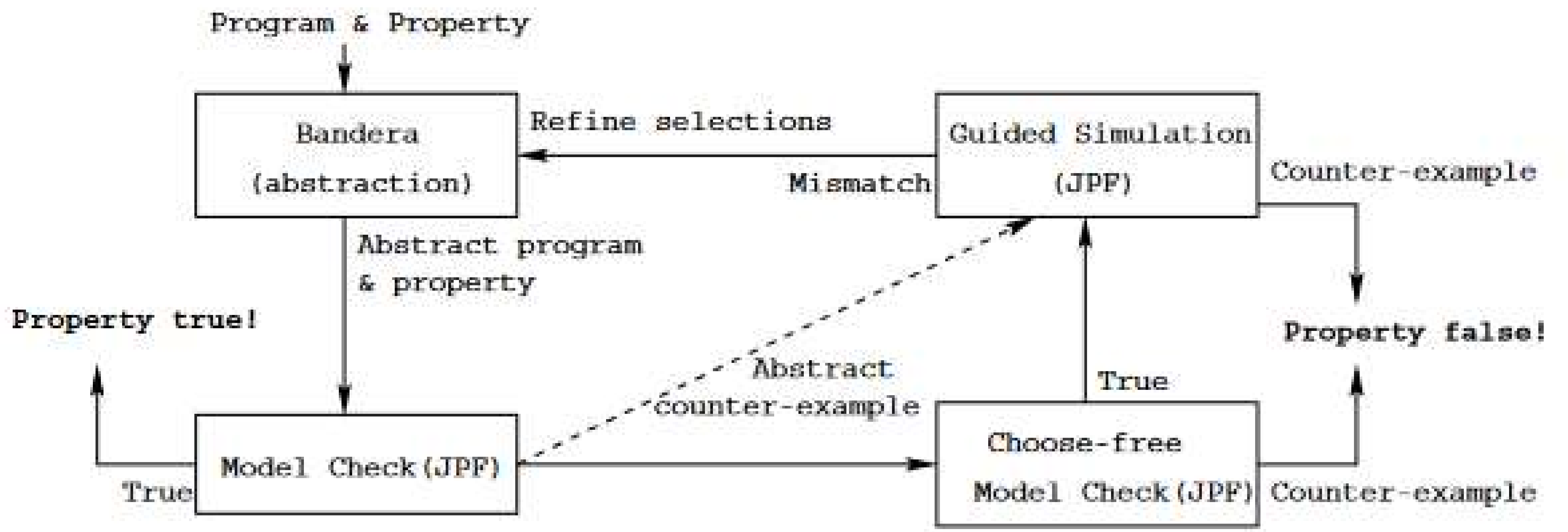

Fig. 4. Model Checking and Refinement

EXAMPLE

```
[1] x=1;                    x=Signs.POS;
[2] y=x+1;                  y=Signs.add(x,Signs.POS);
[3] assert(x<y);            assert((x==Signs.NEG && y==Signs.ZERO)
                                 ||(x==Signs.NEG && y==Signs.POS)
                                 ||(x==Signs.ZERO && y==Signs.POS));
```

Fig. 5. Example of Spurious Error Introduced by Property Abstraction

Source of the error: Counter-example of the abstract model

Techniques like **Choose-Free State Space Search** and **Abstract Counter-Example Guided Concrete Simulation** are designed to mitigate these spurious errors. By focusing on deterministic paths and validating abstract counter-examples through concrete simulation, these techniques help ensure that the counter-examples correspond to real errors in the program.

EVALUATION OVERVIEW

Tested techniques on small to medium-size multi-threaded Java apps

Apps use lock synchronization and condition based synchronization for example wait/notify

RAX

- Remote Agent Experiment
- Component extracted from a spacecraft-control

Pipeline

- Generic multi-threaded staged calculation

RWVSN

- Lea's generic reader-writers synchronization framework
- Uses abstract classes, inheritance and java.util.Vector

DEOS

- Scheduler from a real-time executive for avionics systems.

EVALUATION SETUP

RAX/DEOS ⟶ Examples of these apps had already errors accounted for

Pipeline/RWVSN ⟶ Seeded faults in the program

Program	*SLOC*	*Classes*	*Methods*	*Fields*	*Threads*
RAX	55	4	8	7	3
Pipeline	103	5	10	7	5
RWVSN	590	5	43	10	5
DEOS	1443	20	91	92	6

SLOC - number of source lines of code

EXPERIMENTS - RAX

Without Abstraction

```
class Event{
 int count=0;
 public synchronized void wait_for_event(){
  try{wait();}
  catch(InterruptedException e){};
 }
 public synchronized void signal_event(){
  count = count + 1;
  notifyAll();
 }}
class FirstTask extends Thread{
 Event event1,event2;
 int count=0;
 public void run(){
  count = event1.count;
  while(true){
   if(count == event1.count)
    event1.wait_for_event();
   count = event1.count;
   event2.signal_event();
  }}}
```

With Abstraction

```
class Event {
 int count = Signs.ZERO;
 public synchronized void wait_for_event(){
  try {wait();}
  catch(InterruptedException e){};
 }
 public synchronized void signal_event(){
  count = Signs.add(count,Signs.POS);
  notifyAll();
 }}
class FirstTask extends Thread {
 Event event1,event2;
 int count = Signs.ZERO;
 public void run () {
  count = event1.count;
  while (true){
   if(Signs.eq(count,event1.count))
    event1.wait_for_event();
   count = event1.count;
   event2.signal_event();
  }}}
```

Fig. 6. RAX Program with Deadlock (excerpts)

EXPERIMENTS - PIPELINE

Without Abstraction

```java
class Pipeline {
    private boolean stopCalled = false;
    private int index = 0;

    public synchronized void add() {
        while (index < 2) {
            // Perform pipeline addition
            index++;
        }
    }

    public synchronized void stop() {
        stopCalled = true;
        notifyAll();
    }

    public void stage() {
        assert(stopCalled); // Check that stop was called before terminating stage
    }
}

class PipelineApp {
    public static void main(String[] args) {
        Pipeline pipeline = new Pipeline();

        Thread t1 = new Thread(() -> pipeline.add());
        Thread t2 = new Thread(() -> pipeline.add());

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        pipeline.stop();
        pipeline.stage();

        if (pipeline.stopCalled) {
            System.out.println("Done!");
        }
    }
}
```

With Abstraction

```java
class Pipeline {
    2 references
    boolean stopCalled = false;
    3 references
    int index = Signs.ZERO;

    1 reference
    public synchronized void add() {
        while (Signs.lt(index, Signs.POS)) {
            // Perform pipeline addition
            index = Signs.add(index, Signs.POS);
        }
    }

    1 reference
    public synchronized void stop() {
        stopCalled = true;
        notifyAll();
    }

    1 reference
    public void stage() {
        assert(stopCalled); // Check that stop was called before terminating stage
    }
}

0 references
class PipelineApp {
    3 references
    Pipeline pipeline = new Pipeline();

    0 references
    public void runPipeline() {
        pipeline.add();
        pipeline.stop();
        pipeline.stage();
    }
}
```

EXPERIMENTS - RWVSN

Without Abstraction

```java
class RWVSN {
    public synchronized void readLock() {
    }

    public synchronized void readUnlock() {
        if(assert(!in_writer)) {
            readerCount--;
            notifyAll();
        }
    }

    public synchronized void writeLock() {
        waitingWriters++;
        while (readerCount > 0 || in_writer) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        waitingWriters--;
        writerCount++;
        in_writer = true;
    }

    public synchronized void writeUnlock() {
        writerCount--;
        in_writer = false;
        notifyAll();
    }
}

class RWVSNApp extends Thread {
    private RWVSN rwvsn;

    public RWVSNApp(RWVSN rwvsn) {
        this.rwvsn = rwvsn;
    }

    public void run() {
        rwvsn.writeLock();
        // Perform writing
        rwvsn.writeUnlock();

        rwvsn.readLock();
        // Perform reading
        rwvsn.readUnlock();
    }

    public static void main(String[] args) {
        RWVSN rwvsn = new RWVSN();
        RWVSNApp app1 = new RWVSNApp(rwvsn);
        RWVSNApp app2 = new RWVSNApp(rwvsn);

        app1.start();
        app2.start();
    }
}
```

With Abstraction

```java
class RWVSN {
    int readerCount = Signs.ZERO;
    int writerCount = Signs.ZERO;
    int waitingWriters = Signs.ZERO;
    boolean in_writer = false;

    public synchronized void readLock() {
        while (Signs.gt(writerCount, Signs.ZERO) || Signs.gt(waitingWriters, Signs.ZERO)) {
            try {
                wait();
            } catch (InterruptedException e) {
                // Handle interruption
            }
        }
        readerCount = Signs.add(readerCount, Signs.POS);
    }

    public synchronized void readUnlock() {
        if(!assert(in_writer)){
            readerCount = Signs.sub(readerCount, Signs.POS);
            notifyAll();
        }
    }

    public synchronized void writeLock() {
        waitingWriters = Signs.add(waitingWriters, Signs.POS);
        while (Signs.gt(readerCount, Signs.ZERO) || in_writer) {
            try {
                wait();
            } catch (InterruptedException e) {
                // Handle interruption
            }
        }
        waitingWriters = Signs.sub(waitingWriters, Signs.POS);
        writerCount = Signs.add(writerCount, Signs.POS);
        in_writer = true;
    }

    public synchronized void writeUnlock() {
        writerCount = Signs.sub(writerCount, Signs.POS);
        in_writer = false;
        notifyAll();
    }
}

class RWVSNApp extends Thread {
    RWVSN rwvsn = new RWVSN();

    public void run() {
        rwvsn.writeLock();
        // Perform writing
        assert(!rwvsn.in_writer);
        rwvsn.writeUnlock();

        rwvsn.readLock();
        // Perform reading
        rwvsn.readUnlock();
    }
}
```

EXPERIMENTS - DEOS

Without Abstraction

```java
class DEOS {
    private int field1 = 0;
    private int field2 = 0;
    private int field3 = 0;

    public void schedule() {
        // Example logic involving all three fields
        if (field1 > 0) {
            field2++;
        } else {
            field3++;
        }

        // Example assertion for time partitioning
        assert(field2 != 0 || field3 != 0); // Ensures at least one of the fields has a positive value
    }
}

class DEOSApp {
    private DEOS deos = new DEOS();

    public void runScheduler() {
        deos.schedule();
    }

    public static void main(String[] args) {
        DEOSApp app = new DEOSApp();
        app.runScheduler();
    }
}
```

With Abstraction

```java
class DEOS {
    int field1 = Signs.ZERO;
    int field2 = Signs.ZERO;
    int field3 = Signs.ZERO;

    public void schedule() {
        // Example logic involving all three fields
        if (Signs.gt(field1, Signs.ZERO)) {
            field2 = Signs.add(field2, Signs.POS);
        } else {
            field3 = Signs.add(field3, Signs.POS);
        }

        // Example assertion for time partitioning
        assert(field2 != Signs.ZERO || field3 != Signs.ZERO); // Ensures at least one of the fields has a positive value
    }
}

class DEOSApp {
    DEOS deos = new DEOS();

    public void runScheduler() {
        deos.schedule();
    }
}
```

CONTRIBUTIONS/ACHIEVEMENTS

Authors claim that:

- Counter-example analysis can ease the burden of analyzing abstracted system checks.
- Reduces the length and guarantees feasibility of the counter-examples.
- Choose-free model developed is much faster than the typical model check
- Effective way to exploit more aggressive abstraction since it enables the recovery of feasible counter-examples

RELATED/FUTURE WORK

- Earlier efforts focused on the specification, generation, selection, and compilation of abstractions for Java programs.
- Existing approaches primarily aim at verification, refining abstractions to prove properties, while this work emphasizes defect detection.
- The method used ensures complete coverage of feasible paths, contrasting with approaches that assess single counter-example feasibility.
- Java PathFinder (JPF) leverages a correspondence between concrete and abstracted programs, exploiting Java's default initial values.

RELATED/FUTURE WORK CONT.

- Techniques like forward symbolic simulation and tools like SLAM and INVEST use different methods (e.g., symbolic execution, theorem proving) but are complementary to this work.
- The integration of methods like backward analysis can refine abstractions, enhancing the current techniques for better accuracy and efficiency.

```
    class App{
     public static void main(...){
[1]   new AThread().start(); ...
[2]   int i=0;
[3]   while(i<2){...
[4]     assert(!Global.done);
[5]     i++;
     }}}
    class AThread extends Thread{
     public void run(){ ...
[6]   Global.done=true;
     }}
```

```
class App{
 public static void main(...){
  new AThread().start(); ...
  int i=Signs.ZERO;
  while(Signs.lt(i,Signs.POS)){...
   assert(!Global.done);
   i=Signs.add(i,Signs.POS);
  }}}
class AThread extends Thread{
 public void run(){ ...
  Global.done=true;
 }}
```

Fig. 3. Simple Example of Concrete (left) and Abstracted (right) Code

RELATED/FUTURE WORK CONT.

- Techniques can be applied to any explicit-state model
- The work confirms that theres a path through the abstraction code that is feasible by using the techniques provided by the authors

CONCLUSION

- The paper suggests two approaches for analyzing counter examples produced by model checks of abstracted programs.
- They are fast and are capable of discovering feasible counter-examples in almost every case
- Enabling argressive abstractions without losing error detection
- Treats also thread scheduling policies and property checking
- The techniques can be combined with other counter-examples analysis methods to enhance their precision.