Soundness and Completeness
Warnings in ESC/Java2

Joe Kiniry, Alan Morkan, and Barry Denby

presented by David Cok
by design, neither sound nor complete

popularity of similar tools growing as (lightweight) static analysis tools become more widely used (e.g., Eclipse & FindBugs)

developer comprehension and confidence are paramount (program safety via programmer safety)

complaints from “soundationalists” drives a desire for “tool honesty” and disclosure
Checking Limitations

- a fast, automatic tool must “cheat”
- many scientific and engineering trade-offs
- several sources of soundness and completeness problems
- Java and JML semantic incompleteness
- unsound verification methodology
- limitations of dependent tools (provers)
- problems with user specifications
Requirements on New Warning Subsystem

- contextually warn the user (in detail) about potential soundness and incompleteness
  - e.g., must take into account the program code, annotations, execution path in tool, and theorem prover in use
- provide “tunable” feedback so as to not overwhelm the user with warnings
- be itself sound and complete
- have no false positives or negatives
Detection Methodology

- manually analyze and classify all soundness and completeness issues
- define a type- and annotation-aware AST pattern match for each issue
- each issue implemented as a single “smart” visitor pattern (separation of concerns)
- customized warning levels, messages, and criticality per issue
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
//@ invariant balance <= maxCredit;
public double balance = 0, maxCredit = 100000;
public static int STANDARD = 1, SILVER = 2, GOLD = 4;
private int accountType = 1;

//@ ensures accountType == 4;
public void goldCard() {
    accountType = 4;
}

//@ requires cost < (maxCredit - balance);
//@ ensures \result == \old(balance + cost);
public double purchase(double cost) {
    return balance + cost;
}

//@ ensures (accountType == GOLD ? 1 : 0);
public /*@ pure @*/ boolean isGoldCard() {
    return accountType | GOLD;
}
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
//@ invariant balance <= maxCredit;
public double balance = 0,
//@ requires cost < (maxCredit - balance);
//@ ensures \result == \old(balance + cost);
public double purchase(double cost)
{ return balance + cost; }
//@ ensures (accountType == GOLD ? 1 : 0);
public /*@ pure @*/ boolean isGoldCard()
{ return accountType | GOLD; }
//@ ensures accountType == 4;
public void goldCard()
{ accountType = 4; }
//@ requires accountType == 1;
//@ ensures accountType == 4;
public void goldCard()
{ accountType = 4; }
}
public class CreditCard {
   //@ invariant balance <= maxCredit;
   public double balance = 0, maxCredit = 100000;
   public static int STANDARD = 1, SILVER = 2, GOLD = 4;
   private int accountType = 1;

   //@ ensures accountType == 4;
   public void goldCard() {
      accountType = 4;
   }

   //@ requires cost < (maxCredit - balance);
   //@ ensures \result == \old(balance + cost);
   public double purchase(double cost) {
      return balance + cost;
   }

   //@ ensures (accountType == GOLD ? 1 : 0);
   public /*@ pure @*/ boolean isGoldCard() {
      return accountType | GOLD;
   }
}
```java
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
```
public class CreditCard {
   //@ invariant balance <= maxCredit;
   public double balance = 0, maxCredit = 100000;
   public static int STANDARD = 1, SILVER = 2, GOLD = 4;
   private int accountType = 1;

   //@ ensures accountType == 4;
   public void goldCard() {
      accountType = 4;
   }

   //@ requires cost < (maxCredit - balance);
   //@ ensures \result == \old(balance + cost);
   public double purchase(double cost) {
      return balance + cost;
   }

   //@ ensures (accountType == GOLD ? 1 : 0);
   public /*@ pure @*/ boolean isGoldCard() {
      return accountType | GOLD;
   }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
Example Warnings

Soundness Warning:
Heuristics for class invariant analysis are not sound.

```java
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
```
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
   //@ invariant balance <= maxCredit;
   public double balance = 0, maxCredit = 100000;
   public static int STANDARD = 1, SILVER = 2, GOLD = 4;
   private int accountType = 1;

   //@ ensures accountType == 4;
   public void goldCard() {
      accountType = 4;
   }

   //@ requires cost < (maxCredit - balance);
   //@ ensures \result == \old(balance + cost);
   public double purchase(double cost) {
      return balance + cost;
   }

   //@ ensures (accountType == GOLD ? 1 : 0);
   public /*@ pure @*/ boolean isGoldCard() {
      return accountType | GOLD;
   }
}
public class CreditCard {
//@ invariant balance <= maxCredit;
public double balance = 0, maxCredit = 100000;
public static int STANDARD = 1, SILVER = 2, GOLD = 4;
private int accountType = 1;

//@ ensures accountType == 4;
public void goldCard() {
    accountType = 4;
}

//@ requires cost < (maxCredit - balance);
//@ ensures \result == \old(balance + cost);
public double purchase(double cost) {
    return balance + cost;
}

//@ ensures (accountType == GOLD ? 1 : 0);
public /*@ pure @*/ boolean isGoldCard() {
    return accountType | GOLD;
}
}
public class CreditCard {
   public double balance = 0, maxCredit = 100000;
   public static int STANDARD = 1, SILVER = 2, GOLD = 4;
   private int accountType = 1;

   //@ ensures accountType == 4;
   public void goldCard() {
      accountType = 4;
   }

   //@ requires cost < (maxCredit - balance);
   //@ ensures \result == \old(balance + cost);
   public double purchase(double cost) {
      return balance + cost;
   }

   //@ ensures (accountType == GOLD ? 1 : 0);
   public /*@ pure @*/ boolean isGoldCard() {
      return accountType | GOLD;
   }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
  //@ invariant balance <= maxCredit;
  public double balance = 0, maxCredit = 100000;
  public static int STANDARD = 1, SILVER = 2, GOLD = 4;
  private int accountType = 1;

  //@ ensures accountType == 4;
  public void goldCard() {
    accountType = 4;
  }

  //@ requires cost < (maxCredit - balance);
  //@ ensures \result == \old(balance + cost);
  public double purchase(double cost) {
    return balance + cost;
  }

  //@ ensures (accountType == GOLD ? 1 : 0);
  public /*@ pure @*/ boolean isGoldCard() {
    return accountType | GOLD;
  }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return accountType | GOLD;
    }
}
public class CreditCard {
    //@ invariant balance <= maxCredit;
    public double balance = 0, maxCredit = 100000;
    public static int STANDARD = 1, SILVER = 2, GOLD = 4;
    private int accountType = 1;

    //@ ensures accountType == 4;
    public void goldCard() {
        accountType = 4;
    }

    //@ requires cost < (maxCredit - balance);
    //@ ensures \result == \old(balance + cost);
    public double purchase(double cost) {
        return balance + cost;
    }

    //@ ensures (accountType == GOLD ? 1 : 0);
    public /*@ pure @*/ boolean isGoldCard() {
        return (accountType | GOLD);
    }
}
Benefits and Drawbacks

- increase user awareness of tool limitations
- no more “creeping toward functional verification”
- increase in user confidence
- possible excess of user feedback
  - leads to user confusion and frustration
- text-based warnings need refinement
- prioritization, graphical feedback, etc.
Future Work

- finish implementation
  - only for default code paths
    - strongest postcondition calculus, loop unrolling and safe loops, simplify
- integration with the ESC/Java2 Eclipse plugin and Mobius Tool
- use theorem proving during analysis
- automatic visitor generation
Thank You!

Questions and Comments?
Extra Slides for Questions
Warning Levels

- three options for warnings
  - standard warning mode
  - verbose warning mode
  - no warnings mode
Examining the AST: The Precondition

MethodDecl
  | ModifierPragmaVector
  /    
BinaryExpr
  /    
  
  .....  BinaryExpr (<)
  /    
  
FieldAccess    ParenExpr
  (cost)       |
        
        BinaryExpr(-)
        /    
        
FieldAccess    FieldAccess
  (maxCredit)   (balance)
The Postcondition

MethodDecl
  | ModifierPragmaVector
  / ExprModifierPragma
  | BinaryExpr (==)
  / ResExpr BinaryExpr(+)
  /  \
NaryExpr FieldAccess
  (\old)  (cost)
  | FieldAccess
  (balance)
The Invariant

\[
\begin{align*}
\Gamma, K &\rightarrow L, L \rightarrow M, K \rightarrow M + \Delta, (K \rightarrow M) \equiv (L \rightarrow M \cdot K \rightarrow L) \\
E &\rightarrow F
\end{align*}
\]

\[\text{BinaryExpr}(\leq)\]

\[\text{FieldAccess}(\text{balance}) \quad \text{FieldAccess}(\text{maxCredit})\]