A Relational Framework for Bounded Program Verification

Greg Dennis • Thesis Defense • June 3, 2009
Bugs, bugs, everywhere . . .

- The consequences vary
  from the annoying . . .
  . . . to the tragic

- The costs are high
  - up to $60 billion annually, or .6% of U.S. GDP [NIST 2002]
  - responsible for up to 35% of system downtime [UCB, Gartner]
Existing Program Analyses

- Testing
  - benefit: low investment brings immediate confidence boost
  - challenge: high confidence very costly, hard to obtain
Existing Program Analyses

- **Theorem Proving**
  - benefit: can achieve proof - very high confidence
  - challenge: requires expertise and lots of time
Lots of Static Analyses

- Pattern-based heuristics
  - FindBugs
  - lint

- Abstraction-based model checkers
  - SLAM
  - BLAST

- Abstract interpretation
  - TVLA
  - Astrée

- Explicit-state model checkers
  - SPIN
  - CMC

- Fully automatic, check a variety of properties
Strong vs Weak Specifications

- Static analysis successful for *weak specifications*
  - very partial properties of correctness
  - e.g. null pointer derefs, API conventions, linked list acyclicity

- Less success for *strong specifications*
  - close to full properties of correctness
  - e.g. the radiotherapy machine delivers the prescribed dose
  - e.g. the winner of the election has the most votes

- Strong specs cannot be decomposed into weaker specs
A New Tradeoff

- moderate investment leads to confidence boost
- fully automated, so no proof
- checks strong specifications
Bounded Verification

- Bound *everything* in the program
  - scope of each type (number of instances that may exist)
  - bitwidth of integers
  - number of loop unrollings

- Not a proof
  - exhaustive search within bound

- “Small-scope hypothesis”
  - bugs usually have small examples
  - consistent with our experience and empirical evaluation
Forge Framework

CForge in development by Toshiba Research
Forge Intermediate Representation
Web Registration in FIR

**domain** User, **domain** String

global id: User → Integer
global email: User → String

local newEmail: String
local newId: Integer
local newUser: User

**proc** register (newEmail) : (newUser)
if newEmail ⊆ User.email
  newUser := ∅
else
  newUser := **new** User
  email := email ⋃ (newUser → newEmail)
  newId := **spec**(newId ∉ User.id)
  id := id ⋃ (newUser → newId)

newUser, id, email := **spec**(newEmail ∉ User.email_{old} ⇒
  (newUser.email = newEmail ∧ newUser.id ∉ User.id_{old}))
Web Registration in FIR

domain User, domain String

global id: User → Integer
global email: User → String

local newEmail: String
local newId: Integer
local newUser: User

proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := φ
  else
    newUser := new User
    email := email ∪ (newUser → newEmail)
    newId := spec(newId ⊈ User.id)
    id := id ∪ (newUser → newId)

newUser, id, email := spec(newEmail ⊈ User.email_old ⇒
  (newUser.email = newEmail ∧ newUser.id ⊈ User.id_old))
Web Registration in FIR

**domain** User, **domain** String

**global** id: User → Integer  
**global** email: User → String

**local** newEmail: String  
**local** newId: Integer  
**local** newUser: User

**proc** register (newEmail): (newUser)
  
  **if** newEmail ⊆ User.email
  
  newUser := Ø

  **else**
  
  newUser := **new** User
  
  email := email ∪ (newUser → newEmail)
  
  newId := **spec** (newId ⊈ User.id)
  
  id := id ∪ (newUser → newId)

newUser, id, email := **spec** (newEmail ⊈ User.email_{old} ⇒
  
  (newUser.email = newEmail ∧ newUser.id ⊈ User.id_{old}))

domains are sets  
relational variables
Web Registration in FIR

- **domain** User, **domain** String

- **global** id: User → Integer
- **global** email: User → String

- **local** newEmail: String
- **local** newId: Integer
- **local** newUser: User

**proc** register (newEmail) : (newUser)

```plaintext
if newEmail ⊆ User.email
  newUser := ∅
else
  newUser := new User
  email := email ∪ (newUser → newEmail)
  newId := spec(newId ⊈ User.id)
  id := id ∪ (newUser → newId)
```

newUser, id, email := spec(newEmail ⊈ User.email_{old} ⇒ (newUser.email = newEmail ∧ newUser.id ⊈ User.id_{old}))
Web Registration in FIR

domain User, domain String

global id: User → Integer
global email: User → String

local newEmail: String
local newId: Integer
local newUser: User

proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := ∅
  else
    newUser := new User
    email := email U (newUser → newEmail)
    newId := spec(newId \∉ User.id)
    id := id U (newUser → newId)

newUser, id, email := spec(newEmail \∉ User.email_{old} ⇒ (newUser.email = newEmail ∧ newUser.id \∉ User.id_{old}))
Web Registration in FIR

domain User, domain String

global id: User → Integer
global email: User → String

local newEmail: String
local newId: Integer
local newUser: User

proc register (newEmail) : (newUser)
if newEmail ⊆ User.email
    newUser := ∅
else
    newUser := new User
    email := email ∪ (newUser → newEmail)
    newId := spec(newId ⊈ User.id)
    id := id ∪ (newUser → newId)

newUser, id, email := spec(newEmail ⊈ User.email_{old} ⇒ (newUser.email = newEmail ∧ newUser.id ⊈ User.id_{old}))

domains are sets
relational variables

proc name (inputs) : (outputs)

relational algebra
-empty set
subset
join
cross product
union
conjunction
implication

specification statements
declarative spec in code
non-deterministic change
old refers to pre-state value
enables modular analysis
Relational Logic

Problem:

\[
\begin{align*}
\text{User}^1, \text{friends}^2, \text{connected}^2 \\
\text{friends} \subseteq \text{User} \rightarrow \text{User} \\
\text{friends} = \sim\text{friends} \\
\forall u: \text{User} \mid u \not\in u.\text{friends} \\
\text{connected} = \Diamond\text{friends} \\
\neg\forall v: \text{User} \mid v \subseteq v.\text{connected}
\end{align*}
\]

Bound:

3 User

Solution:

\[
\begin{align*}
\text{User} & \mapsto \{\langle U1 \rangle, \langle U2 \rangle, \langle U3 \rangle\} \\
\text{friends} & \mapsto \{\langle U1, U2 \rangle, \langle U2, U1 \rangle\} \\
\text{connected} & \mapsto \{\langle U1, U2 \rangle, \langle U1, U1 \rangle, \langle U2, U1 \rangle, \langle U2, U2 \rangle\} \\
v & \mapsto \{\langle U3 \rangle\}
\end{align*}
\]
Bounded Verification Analysis

java method
java spec
java bound
java trace
java coverage
jforge
fir procedure
fir spec
fir bound
fir trace
fir coverage
forge
P(t,t')
ψ(t,t')
P ^ ¬ψ
bound
solution
unsat core
kodkod
coverage
trace
spec
procedure
spec
bound
Symbolic Execution

- FIR translated to relational logic by *symbolic execution*

- Generates *symbolic state* \(<D, P, E>\) at each program point:
  - \(D \in \text{Relational Declaration} = \{r_1, r_2, \ldots\}\)
    - set of declared relations in the logic
  - \(P \in \text{Path Constraint} = \{f_1, f_2, \ldots\}\)
    - set of logic formulas that must be true for a feasible execution
  - \(E \in \text{Symbolic Environment} = \{v_1 \leftrightarrow e_1, v_2 \leftrightarrow e_2, \ldots\}\)
    - maps program variables to logical expressions for their value

- Symbolic state encodes a set of feasible program states
Initial Symbolic State

\[
\begin{align*}
\text{domain User, domain String} \\
\text{global id: User} \rightarrow \text{Integer} \\
\text{global email: User} \rightarrow \text{String} \\
\text{local newEmail: String,} \\
\text{local newId: Integer,} \\
\text{local newUser: User} \\
\text{proc register (newEmail) : (newUser)} \\
\quad \text{if newEmail} \subseteq \text{User.email} \\
\quad \quad \text{newUser} := \emptyset \\
\quad \text{else} \\
\quad \quad \text{newUser} := \text{new User} \\
\quad \quad \text{email} := \text{email} \cup (\text{newUser} \rightarrow \text{newEmail}) \\
\quad \quad \text{newId} := \text{spec(newId} \not\subseteq \text{User.id)} \\
\quad \quad \text{id} := \text{id} \cup (\text{newUser} \rightarrow \text{newId})
\end{align*}
\]

\[
\begin{align*}
\mathcal{D}_0 &= \{\text{Integer, } \text{User}_0, \text{String}_0, \\
& \quad \text{id}_0, \text{email}_0, \text{newEmail}_0\} \\
\mathcal{P}_0 &= \{\text{id}_0 \subseteq \text{User}_0 \rightarrow \text{Integer}, \\
& \quad \text{email}_0 \subseteq \text{User}_0 \rightarrow \text{String}_0, \\
& \quad \text{newEmail}_0 \subseteq \text{String}_0\} \\
\mathcal{E}_0 &= \{\text{User} \rightarrow \text{User}_0, \\
& \quad \text{String} \rightarrow \text{String}_0, \\
& \quad \text{id} \rightarrow \text{id}_0, \\
& \quad \text{email} \rightarrow \text{email}_0, \\
& \quad \text{newEmail} \rightarrow \text{newEmail}_0\}
\end{align*}
\]
newEmail ⊆ User.email

newUser := newUser

email := email ∪ (newUser → newEmail)

newId := spec(newId ⊈ User.id)

id := id ∪ (newUser → newId)

newUser := φ

exit
newEmail ⊆ User.email

newUser := new User

email := email ∪ (newUser → newEmail)

newId := spec(newId \∉ User.id)

id := id ∪ (newUser → newId)

newUser := ∅
newUser := \emptyset

email := email \cup (newUser \rightarrow newEmail)

newld := \text{spec}(newld \not\in \text{User.id})

id := id \cup (newUser \rightarrow newld)

exit
newEmail ⊆ User.email

newUser := new User

email := email ∪ (newUser → newEmail)

newId := spec(newId \not∈ User.id)

id := id ∪ (newUser → newId)

newUser := ∅
newUser := new User

email := email ∪ (newUser → newEmail)

newId := spec(newId ⊈ User.id)

id := id ∪ (newUser → newId)

newUser := φ

where cond = newEmail ⊂ User.email
K = {newUser, email, id, User}

exit
Generating the Logic Problem

• Given a specification of the procedure under analysis:

\[
\text{newUser, id, email} := \text{spec}(\text{newEmail} \not\in \text{User}.\text{email}_{\text{old}} \Rightarrow \\
(\text{newUser}.\text{email} = \text{newEmail} \land \text{newUser}.\text{id} \not\in \text{User}.\text{id}_{\text{old}}))
\]

• Use \(E_0\) and \(E_F\) to form the specification formula \(\psi\):

\[
\psi = \text{newEmail}_0 \not\in \text{User}_0.\text{email}_0 \Rightarrow \\
(\text{newUser}_F.\text{email}_F = \text{newEmail}_0 \land \text{newUser}_F.\text{id}_F \not\in \text{User}_0.\text{id}_0)
\]

• Ask Kodkod to find a binding to relations in \(D_F\) such that

\[
P_F \land \neg \psi
\]

“a feasible execution that violates the specification”
Refinement

domain User, domain String

global id: User → Integer
global email: User → String

local newEmail: String
local newId: Integer
local newUser: User

proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := ∅
  else
    newUser := new User
    email := email ∪ (newUser → newEmail)
    newId := spec(newId ⊈ User.id)
    id := id ∪ (newUser → newId)

newUser, id, email := spec(newEmail ⊈ User.email_{old} ⇒
  (newUser.email = newEmail ∧ newUser.id ⊈ User.id_{old}))
Counterexample Trace

domain User, domain String

global id: User → Integer
global email: User → String

local newEmail: String
local maxId: Integer
local newUser: User

proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := ∅
  else
    newUser := new User
    email := email ∪ (newUser → newEmail)
    newId := {m: User.id | ∀ i: User.id | m ≥ i} + 1
    id := id ∪ (newUser → newId)

initial state
User = {U1, U2}
String = {S1, S2, S3}
id = {<U1, 3>, <U2, -4>}
email = {<U1, S1>, <U2, S2>}
newEmail = {S3}

if newEmail ⊆ User.email
  false

newUser := new User
  newUser = {U3}
  User = {U1, U2, U3}

email := email ∪ newUser → newEmail
  email = {<U1, S1>, <U2, S2>, <U3, S3>}

newId := {m: User.id | ∀ i: User.id | m ≥ i} + 1
  newId = {-4}

id := id ∪ (newUser → newId)
  id = {<U1, 3>, <U2, -4>, <U3, -4>}

final state
newUser = {U3}
User = {U1, U2, U3}
email = {<U1, S1>, <U2, S2>, <U3, S3>}
id = {<U1, 3>, <U2, -4>, <U3, -4>
**code**

```plaintext
global cache : Key → Value
local k: Key, local v: Value

proc putCache(k, v) : ()
  if (#cache < 10)
    cache := cache ∪ k → v
  else
    cache := φ → φ

```

**spec**

```plaintext
cache := spec(#cache ≤ 10 ∧ k → v in cache)

```

**code**

```plaintext
global x, y : Integer

proc swap() : ()
  x := y
  y := x

```

**spec**

```plaintext
x, y := spec(x = y_{old})

```

**code**

```plaintext
local n, exponent, result: Integer

proc cube(n) : (result)
  exponent := 3
  result := 1
  while (exponent > 0)
    result := result × n

```

**spec**

```plaintext
result := spec(result = n × n × n)

```

- **Insufficient bound**
- **Underconstrained specification**
- **Infinite loop**
Coverage Metric
Measuring Coverage

```plaintext
proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := \emptyset;
  else
    newUser := new User;
    email := email ∪ (newUser → newEmail);
    newId := spec(newId ∉ User.id);
    id := id ∪ (newUser → newId);

newUser, id, email := spec(newEmail ∉ User.email_{old} ⇒
  (newUser.email = newEmail ∧ newUser.id ∉ User.id_{old}))
```

\[ F \in \text{Formula Slice Map} \]
- maps formulas to statements

\[ P_f \]

\[ \text{unsatisfiable core} \]
\[ \text{missed statements} \]
newEmail \subseteq User.email

newUser := new User

email := email \cup (newUser \rightarrow newEmail)

newId := spec(newId \not\in User.id)

id := id \cup (newUser \rightarrow newId)

exit

newUser := \emptyset

\langle D_0, P_0, E_0 \rangle

\langle D_0 \cup \{newUser_2\},
P_0 \cup \{newUser_2 = \emptyset\},
E_0[newUser \leftrightarrow newUser_2] \rangle

F_2 = \{((newUser_2 = \emptyset) \mapsto \{S_2\})\}

\langle D_F, P_F, E_F \rangle
Symbolic Execution Strategies

- Two approaches to symbolic execution

- “Inline Strategy”
  - avoids declaring new relations or adding path constraints
  - when possible, encodes statement effect in environment
  - more compact, faster solving times

- “Constrain Strategy”
  - binds modified variables to fresh relations
  - constrains value of fresh relations in path constraint
  - leaves formula “breadcrumbs” to calculate coverage
newUser := new User

newEmail ⊆ User.email

email := email ∪ (newUser → newEmail)

newId := spec(newId \notin User.id)

id := id ∪ (newUser → newId)

newUser := ∅
newEmail \subseteq \text{User.email}

newUser := \textbf{new User}

email := email \cup (\text{newUser} \rightarrow \text{newEmail})

cond \Rightarrow f_2, \\
\neg \text{cond} \Rightarrow f_3, \\
\neg \text{cond} \Rightarrow f_4, \\
\neg \text{cond} \Rightarrow f_5, \\
\neg \text{cond} \Rightarrow f_6

F_F = \{(\text{cond} \Rightarrow f_2) \mapsto \{S_1, S_2\}, \\
(\neg \text{cond} \Rightarrow f_3) \mapsto \{S_1, S_3\}, \\
(\neg \text{cond} \Rightarrow f_4) \mapsto \{S_1, S_4\}, \\
(\neg \text{cond} \Rightarrow f_5) \mapsto \{S_1, S_5\}, \\
(\neg \text{cond} \Rightarrow f_6) \mapsto \{S_1, S_6\}\}

F_6 = \{f_3 \mapsto \{S_3\}, f_4 \mapsto \{S_4\}, f_5 \mapsto \{S_5\}, f_6 \mapsto \{S_6\}\}
Measuring Coverage

```plaintext
proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := Ø;
  else
    newUser := new User;
    email := email ∪ (newUser → newEmail);
    newId := spec(newId ∉ User.id);
    id := id ∪ (newUser → newId);

newUser, id, email := spec(newEmail ∉ User.email_{old} ⇒ (newUser.email = newEmail ∧ newUser.id ∉ User.id_{old}))

F_F = {
  cond ⇒ f_2 ⇔ {S_1, S_2},
  ¬cond ⇒ f_3 ⇔ {S_1, S_3},
  ¬cond ⇒ f_4 ⇔ {S_1, S_4},
  ¬cond ⇒ f_5 ⇔ {S_1, S_5},
  ¬cond ⇒ f_6 ⇔ {S_1, S_6},
}
¬ψ
```

= unsatisfiable core
= missed statements
Small-Scope Evaluation

10 benchmarks

MuJava

429 mutants

- 46 equivalent
- 16 infinite
- 367 killable

<table>
<thead>
<tr>
<th>benchmark</th>
<th>killable</th>
<th>min bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>b0</td>
<td>62</td>
<td>1s 3b 2u</td>
</tr>
<tr>
<td>b1</td>
<td>18</td>
<td>1s 2b 2u</td>
</tr>
<tr>
<td>b2</td>
<td>20</td>
<td>1s 2b 1u</td>
</tr>
<tr>
<td>b3</td>
<td>8</td>
<td>2s 2b 1u</td>
</tr>
<tr>
<td>b4</td>
<td>14</td>
<td>2s 4b 1u</td>
</tr>
<tr>
<td>b5</td>
<td>57</td>
<td>1s 3b 3u</td>
</tr>
<tr>
<td>b6</td>
<td>61</td>
<td>0s 3b 2u</td>
</tr>
<tr>
<td>b7</td>
<td>4</td>
<td>2s 2b 1u</td>
</tr>
<tr>
<td>b8</td>
<td>80</td>
<td>1s 4b 3u</td>
</tr>
<tr>
<td>b9</td>
<td>43</td>
<td>3s 2b 1u</td>
</tr>
</tbody>
</table>

367 killable

chart with percentage of mutants killed vs. bound (scope=bitwidth=unrollings)
Coverage Evaluation

- How well does the metric detect infinite loops?
  - Does it find statements to be missed in the infinite mutants?
  - Yes, in all 16
  - Example:

```
static void insertion(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        int B = a[i];
        while ((j > 0) && (a[j - 1] > B)) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = B;
    }
}
```

- How well does the metric detect insufficient bounds?

<table>
<thead>
<tr>
<th>bound</th>
<th>survived</th>
<th>detected</th>
<th>% detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope - 1</td>
<td>260</td>
<td>260</td>
<td>100%</td>
</tr>
<tr>
<td>bitwidth - 1</td>
<td>135</td>
<td>118</td>
<td>87%</td>
</tr>
<tr>
<td>unrollings - 1</td>
<td>27</td>
<td>24</td>
<td>89%</td>
</tr>
<tr>
<td></td>
<td>422</td>
<td>402</td>
<td>95%</td>
</tr>
</tbody>
</table>
Case Study: KOA Voting Software

- Kiezen Op Afstand ("Remote Voting")
  - first open source internet voting application
  - 2004 pilot program for overseas voters in the Netherlands

- Vote-Tallying Subsystem
  - built by Security of Systems (SoS) research group
  - SoS group are the developers of ESC/Java2
  - formal methods pledged in their winning bid
Rigorous Development

- Specifications in Java Modeling Language (JML)
- Checked with ESC/Java2 and tested with JMLUnit

```java
public class VoteSet {

    private /*@ spec_public @*/ boolean my_vote_has_been_initialized; //@ in objectState;
    private /*@ spec_public @*/ boolean my_vote_has_been_finalized; //@ in objectState;
    //@ invariant my_vote_has_been_finalized ==> my_vote_has_been_initialized;

    /**
    * <pre><jml>
    * normal_behavior
    *   requires !my_vote_has_been_initialized && !my_vote_has_been_finalized;
    *   assignable my_vote_has_been Initialized;
    *   ensures my_vote_has_been_initialized;
    * also
    *   exceptional_behavior
    *   requires my_vote_has_been_initialized || my_vote_has_been_finalized;
    *   assignable \nothing;
    *   signals (IllegalArgumentException) true;
    * </jml></pre>
    */
    
    final void initializeVote() throws IllegalArgumentException {
        if (my_vote_has_been_initialized || my_vote_has_been_finalized) {
            throw new IllegalArgumentException();
        }
        my_vote_has_been_initialized = true;
    }
}
```
Forge Results on KOA

- starting bound: scope=5, bitwidth=4, unrollings=3
- on timeout, lowered scope and re-analyzed

<table>
<thead>
<tr>
<th>class</th>
<th>methods</th>
<th>violations</th>
<th>mean scope</th>
<th>mean time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuditLog</td>
<td>90</td>
<td>1</td>
<td>5</td>
<td>2.3</td>
</tr>
<tr>
<td>District</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>18.5</td>
</tr>
<tr>
<td>CandidateListMetadata</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>33.6</td>
</tr>
<tr>
<td>Candidate</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>59.3</td>
</tr>
<tr>
<td>KiesLijst</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>104.6</td>
</tr>
<tr>
<td>KiesKring</td>
<td>15</td>
<td>5</td>
<td>5</td>
<td>249.7</td>
</tr>
<tr>
<td>CandidateList</td>
<td>13</td>
<td>3</td>
<td>4.5</td>
<td>1416.8</td>
</tr>
<tr>
<td>VoteSet</td>
<td>11</td>
<td>4</td>
<td>3.7</td>
<td>1783.9</td>
</tr>
<tr>
<td><strong>Sum or Mean</strong></td>
<td><strong>169</strong></td>
<td><strong>19</strong></td>
<td><strong>4.9</strong></td>
<td><strong>262.7</strong></td>
</tr>
</tbody>
</table>
Counterexample Trace

**initial state:**
```
this = {KiesKring0}
my_district_count = {<KiesKring0, 2>, <KiesKring1, 1>}
```

**Stmt17:**
```
c := this.my_district_count goto Stmt19
c = {2}
```
private final /*@ non_null @*/ District[] my_districts;
private byte my_district_count;

//@ invariant my_district_count ==
//@  (\sum int i; 0 <= i && i < my_districts.length;
//@    (my_districts[i] != null) ? 1 : 0);

boolean addDistrict(final /*@ non_null @*/ District a_district) {
    if (hasDistrict(a_district)) {
        return false;
    }
    my_districts[a_district.number()] = a_district;
    my_district_count++;
    return true;
}
private final /*@ non_null @*/ District[] my_districts
private byte my_district_count;

//@ invariant my_district_count ==
//@  (∑ int i; 0 <= i && i < my_districts.length;
//@           (my_districts[i] != null) ? 1 : 0);

boolean addDistrict(final /*@ non_null @*/ District a_district) {
  if (hasDistrict(a_district)) {
    return false;
  }
  my_districts[a_district.number()] = a_district;
  my_district_count++;
  return true;
}

• hasDistrict returns true when argument has the same number and name as existing district
• overwrites array and increments count when argument has same number and different name
• unlikely to find with testing, even with 100% coverage
VoteSet.addVote [spec bug]

class VoteSet {

    final void addVote(final int a_candidate_code) throws IllegalArgumentException {
        if (!(my_vote_has_been_initialized && !my_vote_has_been_finalized)) {
            throw new IllegalArgumentException();
        }
        final Candidate candidate = my_candidate_list.getCandidate(a_candidate_code);
        candidate.incrementVoteCount();
        candidate.kiesLijst().incrementVoteCount();
    }
}

class Candidate {

    //@ requires my_vote_count < AuditLog.getDecryptNrOfVotes();
    //@ modifies my_vote_count;
    //@ ensures my_vote_count == \old(my_vote_count + 1); final
    int incrementVoteCount() { . . . }
}
VoteSet.addVote [spec bug]

class VoteSet {
    final void addVote(final int a_candidate_code) throws IllegalArgumentException {
        if (!(my_vote_has_been_initialized && !my_vote_has_been_finalized)) {
            throw new IllegalArgumentException();
        }
        final Candidate candidate = my_candidate_list.getCandidate(a_candidate_code);
        candidate.incrementVoteCount();
        candidate.kiesLijst().incrementVoteCount();
    }
}

class Candidate {
    //@ requires my_vote_count < AuditLog.getDecryptNrOfVotes();
    //@ modifies my_vote_count;
    //@ ensures my_vote_count == \old(my_vote_count) + 1; final
    int incrementVoteCount() { . . . }
}

- addVote does not include precondition of incrementVoteCount
- when fixed do dependent methods break?
# Specification Violations

19 violations detected

- 3 code bugs
- 16 spec bugs

lowered bound to find minimal that reveals bug

<table>
<thead>
<tr>
<th>class</th>
<th>method</th>
<th>bug</th>
<th>min bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>CandidateListMetadata</td>
<td>init</td>
<td>spec</td>
<td>1s 3b 1u</td>
</tr>
<tr>
<td>KiesKring</td>
<td>addDistrict</td>
<td>code</td>
<td>1s 3b 1u</td>
</tr>
<tr>
<td>VoteSet</td>
<td>addVote(String)</td>
<td>spec</td>
<td>1s 3b 1u</td>
</tr>
<tr>
<td>KiesLijst</td>
<td>clear</td>
<td>spec</td>
<td>1s 3b 3u</td>
</tr>
<tr>
<td>AuditLog</td>
<td>getCurrentTimeStamp</td>
<td>spec</td>
<td>2s 1b 1u</td>
</tr>
<tr>
<td>Candidate</td>
<td>init</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>CandidateList</td>
<td>addDistrict</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>CandidateList</td>
<td>addKiesLijst</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>CandidateList</td>
<td>init</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesKring</td>
<td>addKiesLijst</td>
<td>code</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesKring</td>
<td>init</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesKring</td>
<td>make</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesLijst</td>
<td>addCandidate</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesLijst</td>
<td>compareTo</td>
<td>code</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesLijst</td>
<td>make</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>VoteSet</td>
<td>addVote(int)</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>VoteSet</td>
<td>validateKiesKringNumber</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>VoteSet</td>
<td>validateRedundantInfo</td>
<td>spec</td>
<td>2s 3b 1u</td>
</tr>
<tr>
<td>KiesKring</td>
<td>clear</td>
<td>spec</td>
<td>2s 3b 3u</td>
</tr>
</tbody>
</table>
Discussion

• Bounded verification is useful
  - found problems in code that had been rigorously analyzed
  - more support for the small-scope hypothesis
  - coverage metric mitigates unsoundness
  - moderate cost + automation + strong specifications

• Comparison to ESC/Java2
  - decision procedures with unsound heuristics
    ✅ if no heuristics needed, can obtain a proof
    🚫 misses proofs in practice, no counterexample traces

• Comparison to Java PathFinder
  - explicit-state model checking with unsound heuristics
    ✅ no specifications of called methods, concurrent properties
    🚫 applicable only to closed systems, assertion specs are limiting
Future

- **Concern: ease of use**
  - better debugger-like UI for stepping through traces
  - relational specification language for Java [Yessenov]
  - infer specifications of called procedures [Taghdiri]

- **Concern: scalability**
  - exploit generics in source code to reduce state space
  - logarithmic encoding of integers [Vaziri + Dolby]
  - symmetry breaking for acyclic fields [Galeotti]

- **Concern: soundness**
  - use decision procedures for decidable portions
  - use bounding for undecidable portions
  - ESC/Forge hybrid?
Vision

- Give Forge a try!
  - http://sdg.csail.mit.edu/forge/

unsound verification?  formal testing?

testing

ESC/Java2
Forge
Kiasan
JPF-SE

cost

confidence

theorem proving