Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries

Cormac Flanagan
University of California, Santa Cruz
Software Reliability

This program has performed an illegal operation. If the problem persists, please contact the vendor.

Physical memory available to Windows: 163,336 KB

Lecture 0
Summ 0
1 Sep 2003, Tunis, Tunisia
Software Reliability

- Testing
  - dominant methodology
  - costly
  - test coverage problem

- Static checking
  - combats test coverage by considering all paths
  - Type systems
    - very effective for certain type errors
  - Extended Static Checking
    - targets errors missed by type systems
The translator “understands” the semantics of Java.

An error condition is a logical formula (boolean combination of constraints) that, ideally, is satisfiable if and only if the program contains an error.

The satisfiability checker is invisible to users.

Satisfying assignments are turned into precise warning messages.
ESC/Java Example

class Rational {
    //@ invariant den != 0;
    int num, den;

    //@ requires d != 0;
    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        return num/den;
    }
}

d public static void main(String[] a) {
    int n = readInt(), d = readInt();
    if( d == 0 ) return;
    Rational r = new Rational(d,n);
    for(int i=0; i<10000; i++) {
        print( r.trunc() );
    }
}
ESC/Java Experience

- Tested on 40+ KLOC (Java front end, web crawler, ...)
- Finds software defects!
- Useful educational tool
- Annotation cost significant
  - 100 annotations per KLOC
  - 3 programmer-hours per KLOC
- Annotation overhead significantly limits widespread use of extended static checking
- Why do we need these annotations?
The translator “understands” the semantics of Java.

An error condition is a logical formula that, ideally, is satisfiable if and only if the program contains an error.

The satisfiability checker is invisible to users.

Satisfying assignments are turned into precise warning messages.

Index out of bounds on line 218
Method does not preserve object invariant on line 223
Generating Error Conditions

Java program + Annotations

ESC/Java
Translator

Error conditions

Satisfiability checker
Satisfying assignment

Post-processor

Warning messages

if (x < 0) { x := -x; }
//@ assert x >= 0;

( x<0 ∧ x'=-x ∧ ¬(x'>=0) )
∨ (¬(x<0) ∧ ¬(x>=0) )

Unsatisfiable, no error
Generating Error Conditions 2

Java program + Annotations

ESC/Java

Translator

Error conditions

Satisfiability checker

Satisfying assignment

Post-processor

Warning messages

\[
\begin{align*}
p & := q; \\
p.f & := 3; \\
//@ assert q.f != 0; \\
p &= q \\
\wedge f' &= \text{store}(f, p, 3) \\
\wedge \text{select}(f', q) &= 0\\
\text{Unsatisfiable, no error}
\end{align*}
\]
ESC/Java Error Condition Logic

Java program + Annotations

- terms \( t ::= x \mid f(t) \)
- constraints \( c ::= p(t) \)
- formulae \( e ::= c \mid \neg c \mid e \land e \mid e \lor e \mid \exists y. e \)
- theories: equality, linear arithmetic, select+store
- some heuristics for universal quantification
- logic cannot express iteration or recursion
Error Conditions for Procedures

Java program + Annotations

- Translator
- Error conditions
- Satisfiability checker
- Satisfying assignment
- Post-processor
- Warning messages

- call p(x);
  assert x>0
- //@ requires pre(x)
  //@ ensures post(x,x')
  void p(x) { ... }
- \(\neg pre(x)\)
- \(\lor \exists x'. (post(x,x') \land \neg (x'>0))\)

- Programmer must write procedure specs
- so procedure calls can be translated away
- because EC logic cannot express recursion

- Many warnings about incorrect or incomplete specifications
Verifun Architecture

Procedure definitions and calls in source program are translated into analogous relation definitions and calls in logic
- no need for procedure specifications

Constraint Logic Programming (CLP) query
- can express relation definitions and calls

Satisfiability checker for CLP

The error message includes a complete execution trace
- no warnings about missing specifications
Verifun Architecture

Java program (w/ calls)

German

• terms \( t ::= x \mid f(t) \)
• constraints \( c ::= p(t) \)
• formulas \( e ::= c \mid \neg c \mid e \land e \mid e \lor e \mid \exists y. e \mid r(t) \)
• user-defined relation symbols \( r \)
• definitions \( d ::= r(x) :- e \)

• Query: Given \( d \), is \( e \) satisfiable?

• Constraint Logic Programming
  - [Jaffar and Lassez, POPL’87]
  - Efficient implementations! 😊
Error Conditions for Procedures

Java program (w/ calls) →

- call p(x);
- assert x>0

void p(x) { S }

\[ Ep(x) \lor \exists x'. (Tp(x,x') \land \neg(x'>0)) \]

- \( ec(S, R) \) describes states from which \( S \) fails an assertion or terminates in state satisfying \( R \)
- error relation true if \( p \) goes wrong from \( x \)
  - \( Ep(x) :- ec(S, false) \)
- transfer relation relates pre, post values of \( x \)
  - \( Tp(x,x') :- ec(S, (x=x')) \)
Example: Factorial

```c
int fact(int i) {
    if (i == 0) return 1;
    int t = fact(i-1);
    assert t > 0;
    return i*t;
}

void main() {
    int j = readInt();
    fact(j);
}
```

```prolog
Tfact(i, r) :-
    ( i = 0 ∧ r = 1)
    ∨ ( i != 0 ∧ Tfact(i-1, t)
        ∧ t>0 ∧ r=i*t   )

Efact(i) :-
    i != 0
    ∧ ( Efact(i-1)
        ∨ (Tfact(i-1, t) ∧ t <= 0 ))

Emain() :- Efact(j)
```

- Tfact(i, r) relates pre and post states of executions of fact
- Efact(i) describes pre-states from which fact may go wrong
- CLP has least fixpoint semantics
- CLP Query: Is Emain() satisfiable?
**Imperative Software**

- Program correctness
- Bounded software model checking

**Constraint Logic Programming**

- CLP satisfiability
- Efficient implementations
  - Sicstus Prolog (depth-first)
Example: Rational

class Rational {

    int num, den;

    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        return num/den;
    }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if( d == 0 ) return;
        Rational r = new Rational(d,n);
        for(int i=0; i<10000; i++) {
            print( r.trunc() );
        } } }
Error Condition for Rational

t_rat(AllocPtr, Num, Den, N, D, This, AllocPtrp, Nump, Denp) :-
    AllocPtrp is AllocPtr+1,
    This = AllocPtrp,
    aset(This,Den,D,Denp),
    aset(This,Num,N,Nump).

t_readInt(R).

e_trunc(This, Num, Den) :- aref(This,Den,D), \{D = 0\}.

e_loop(I, This, Num, Den) :- e_trunc(This, Num, Den).

**e_loop(I, This, Num, Den) :- \{I<10000, Ip=I+1\}, e_loop(Ip,This, Num, Den).**

**e_main :-**
    AllocPtr = 0, ;; no objs allocated
    new_array(Num), ;; initialise arrays encoding fields Num
    new_array(Den), ;; and Den
    t_readInt(D),
    t_readInt(N),
    \{D =\= 0\},
    t_rat(AllocPtr, Num, Den, D, N, This, AllocPtrp, Nump, Denp),
    e_loop(0, This, Nump, Denp).
Checking Rational with CLP

- Feed error condition into CLP implementation (SICStus Prolog)
  - *explicitly* explores all paths through EC
  - *symbolically* considers all data values, eg, for n,d
    - using theory of linear arithmetic
  - quickly finds that the EC is satisfiable
    - gives satisfying derivation for EC
    - can convert to erroneous execution trace
Example: Fixed Rational

class Rational {

    int num, den;

    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        return num/den;
    }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if( d == 0 ) return;
        Rational r = new Rational(n,d);
        for(int i=0; i<10000; i++) {
            print( r.trunc() );
        }
    }
}
Checking Fixed Rational with CLP

- Feed error condition into CLP implementation
  - *explicitly* explores all paths through EC
  - *symbolically* considers all data values, eg, for $n,d$
  - but searches through all possible program paths
    - 10,000 iterations of loop
    - depth-first search
    - before saying EC is unsatisfiable and program is ok

- Programs typically have infinitely many paths
  - Standard CLP checkers may not terminate
Deciding CLP Queries: Avoiding All Paths

- Use ideas from verification/model checking to decide CLP queries without exploring all paths
- Explicating theorem proving
- Predicate abstraction + predicate inference
  - [SLAM, BLAST]
Review: Explicating Theorem Proving

Query $Q^b$:
\[
\land !\{d = 0\} \\
\land \lor \{\text{num}' = \text{store}(\text{num}, \text{this}, n)\} \\
\land \lor \{\text{num}' = \text{store}(\text{num}, \text{this}, 0)\} \\
\land \{\text{den}' = \text{store}(\text{den}, \text{this}, d)\} \\
\land \{\text{select}(\text{den}', \text{this}) = 0\}
\]

Truth assignment $TA$:
\[
!(d = 0) \\
\text{num}' = \text{store}(\text{num}, \text{this}, n) \\
\text{den}' = \text{store}(\text{den}, \text{this}, d) \\
\text{select}(\text{den}', \text{this}) = 0
\]

//@ invariant den != 0
//@ requires d != 0
Rational(int n, int d) {
    if (*) num = n;
    else num = 0;
    den = d;
}

Unsatisfiable!
Explicating Refinement for CLP Queries

• Use the same approach
  - abstract to a decidable boolean system
  - get “trace” from boolean system
  - check trace using proof-generating decision procedures
  - if trace is invalid, use invalidity proof to refine the boolean abstraction

• But boolean abstraction must now include relations
Abstracting CLP Relations

• Abstract each relation definition \( r(x) :- e \)

• to *propositional* relation definition \( R(B) :- E \)
  - \( B \) is a list of boolean variables
  - \( E \) is a propositional formula
  - satisfiability is decidable
## Abstraction Refinement for CLP - 1

<table>
<thead>
<tr>
<th>Program:</th>
<th>Error Condition Q:</th>
<th>Abstract EC Q^α:</th>
</tr>
</thead>
</table>
| void inc(x) {  
  int y = x;  
  x=(* ? 0 : y+1);  
} | Tinc(x,x') :-  
  ∧ y=x  
  ∧ ∨ x'=0  
  ∨ x'=y+1 | TINC() :-  
  ∧ {y=x}  
  ∧ ∨ {x'=0}  
  ∨ {x'=y+1} |
| void main() {  
  int z = 0;  
  inc(z);  
  assert z = 1;  
} | Emain() :-  
  ∧ z=0  
  ∧ Tinc(z,z')  
  ∧ ¬(z'=1) | EMAIN() :-  
  ∧ {z=0}  
  ∧ TINC()  
  ∧ ¬{z'=1} |

Is Emain() satisfiable?  
Is EMAIN() sat?
### Abstraction Refinement for CLP - 2

**Abstract EC Q^a:**

- **TINC() :-**
  - \( \wedge \{y=x\} \)
  - \( \wedge \vee \{x'=0\} \)
  - \( \vee \{x'=y+1\} \)

- **EMAIN() :-**
  - \( \wedge \{z=0\} \)
  - \( \wedge \neg \{z'=1\} \)

Is **EMAIN()** sat?  

**Abstract Trace T^a:**

- **TINC() :-**
  - \( \wedge \{y=x\} \)
  - \( \wedge \{x'=y+1\} \)

- **EMAIN() :-**
  - \( \wedge \{z=0\} \)
  - \( \wedge \neg \{z'=1\} \)

**Concrete Trace T:**

- **Tinc(x,x') :-**
  - \( \wedge y=x \)
  - \( \wedge x'=y+1 \)

- **Emain() :-**
  - \( \wedge z=0 \)
  - \( \wedge Tinc(z,z') \)
  - \( \neg (z'=1) \)

**EMAIN() is sat.**
## Abstraction Refinement for CLP - 3

<table>
<thead>
<tr>
<th>Concrete Trace $T$:</th>
<th>Conjunction of Atoms:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Tinc}(x,x')$ :-</td>
<td>$\wedge y=x$</td>
</tr>
<tr>
<td>$\wedge y=x$</td>
<td>$\wedge x'=y+1$</td>
</tr>
<tr>
<td>$\wedge x'=y+1$</td>
<td></td>
</tr>
</tbody>
</table>

| $\text{Emain}()$ :- | $\wedge z=0$ |
| $\wedge z=0$ | $\wedge z=x \wedge z'=x'$ |
| $\wedge \text{Tinc}(z,z')$ | $\wedge -(z'=1)$ |
| $\wedge -(z'=1)$ | |
**Abstraction Refinement for CLP - 4**

**Conjunction of Atoms:**

\[\land y=x\land x'=y+1\land z=0\land z=x\land z'=x'\land \neg(z'=1)\]

**Explicated clause in Emain()**

\[z'=z+1 \land z=0 \land \neg(z'=1) \Rightarrow \text{false}\]

**Explicated clause in Tinc()**

\[y=x \land x'=y+1 \Rightarrow x'=x+1\]

**Argument binding inference**

Add \(x'=x+1\) as interface predicate for Tinc()
Abstraction Refinement for CLP - 5

Error Condition $Q$:

\[
\text{Tinc}(x,x') \leftarrow \\
\quad \land y = x \\
\quad \lor x' = 0 \\
\quad \lor x' = y + 1
\]

\[
\text{Emain()} \leftarrow \\
\quad \land z = 0 \\
\quad \land \text{Tinc}(z,z') \\
\quad \land \neg (z' = 1)
\]

Abstract EC $Q^a$:

\[
\text{TINC}\{x'=x+1\} \leftarrow \\
\quad \land \{y=x\} \\
\quad \lor \{x'=0\} \\
\quad \lor \{x'=y+1\} \\
\quad \land (\{y=x\} \land \{x'=y+1\} \Rightarrow \{x'=x+1\})
\]

\[
\text{EMAIN()} \leftarrow \\
\quad \land \{z=0\} \\
\quad \land \text{TINC}\{z'=z+1\} \\
\quad \land \neg \{z'=1\} \\
\quad \land \neg (\{z'=z+1\} \land \{z=0\} \land \neg \{z'=1\})
\]
Abstraction Refinement for CLP - 6

Abstract EC Qa:

\[ \text{TINC}({x'=x+1}) :- \]
\[ \land \{y=x\} \]
\[ \land \lor \{x'=0\} \]
\[ \lor \{x'=y+1\} \]
\[ \land (\{y=x\} \land \{x'=y+1\} \Rightarrow \{x'=x+1\}) \]

\[ \text{EMAIN}() :- \]
\[ \land \{z=0\} \]
\[ \land \text{TINC}({z'=z+1}) \]
\[ \land \neg\{z'=1\} \]
\[ \land \neg(\{z'=z+1\} \land \{z=0\} \land \neg\{z'=1\}) \]

Abstract Trace Ta:

\[ \text{TINC}({x'=x+1}) :- \]
\[ \land \{y=x\} \]
\[ \land \{x'=0\} \]
\[ \land \neg\{x'=y+1\} \]
\[ \land \neg\{x'=x+1\} \]

\[ \text{EMAIN}() :- \]
\[ \land \{z=0\} \]
\[ \land \text{TINC}({z'=z+1}) \]
\[ \land \neg\{z'=1\} \]
\[ \land \neg\{z'=z+1\} \]
**Abstract Trace** \( T^a \):

\[
\text{TINC(}\{x'=x+1\} \text{)} :- \\
\land \{y=x\} \\
\land \{x'=0\} \\
\land \neg \{x'=y+1\} \\
\land \neg \{x'=x+1\}
\]

\[
\text{EMAIN()} :- \\
\land \{z=0\} \\
\land \text{TINC(}\{z'=z+1\}\text{)} \\
\land \neg \{z'=1\} \\
\land \neg \{z'=z+1\}
\]

**Concrete Trace** \( T \):

\[
\text{TINC(x,x)} :- \\
\land y=x \\
\land x'=0 \\
\land \neg (x'=y+1) \\
\land \neg (x'=x+1)
\]

\[
\text{EMAIN()} :- \\
\land z=0 \\
\land \text{TINC(}z,z'\text{)} \\
\land \neg (z'=1) \\
\land \neg (z'=z+1)
\]

**Conjunction of Constraints**

\[
\land y=x \\
\land x'=0 \\
\land \neg (x'=y+1) \\
\land \neg (x'=x+1)
\]
Abstraction Refinement for CLP - 7

Conjunction of Constraints:

\[ \land y = x \]
\[ \land x' = 0 \]
\[ \land \neg (x' = y + 1) \]
\[ \land \neg (x' = x + 1) \]
\[ \land z = 0 \]
\[ \land z = x \land z' = x' \]
\[ \land \neg (z' = 1) \]
\[ \land \neg (z' = z + 1) \]

Program Trace:

```c
void inc(x) {
    int y = x;
    x = (* ? 0 : y + 1);
}

void main() {
    int z = 0;
    inc(z);
    assert z = 1;
}
```

decision procedures say this conjunction of constraints is consistent, so this is a real error trace
Verifun on Fixed Rational

class Rational {
    int num, den;

    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        return num/den;
    }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if( d == 0 ) return;
        Rational r = new Rational(n,d);
        for(int i=0; i<10000; i++) {
            print( r.trunc() );
        }
    }
}

Abstract EC unsatisfiable
Program correct

Interface predicate:
    den'=store(den,this,d)

Interface predicate:
    select(den,this)=0

Explicated clause:
    den'=store(den,r,d)
    ∧ select(den',r)=0
    ⇒ d = 0
Properties of Verifun CLP Checker

- **Sound**
  - only produces satisfying traces
- **Complete**
  - will find a satisfying trace, if one exists
- **Progress**
  - never considers the same trace twice
- **Semi-algorithm**
  - termination depends on discovering sufficient predicates
  - may not terminate
  - could bound depth of recursion
Unit of Specification

- Programmers work on “unit of development”
- Interfaces between such units must be specified
  - reasonable to make specifications formal
- Use Verifun to check unit of development with respect to its specification

- Limitation of ESC is that the unit of specification (procedure) is much smaller than unit of development
Imperative Software

- Program correctness
- Bounded software model checking
- Explicating theorem proving
- Predicate abstraction & predicate inference
  - SLAM, BLAST

Constraint Logic Programming

- CLP satisfiability
- Efficient implementations
  - Sicstus Prolog
  - depth-first
- CLP implementation technique
  - Avoids considering all paths
  - Verifun CLP satisfiability checker
Related Work

• Program checking
  - Stanford Pascal Verifier, ESC/M3, ESC/Java
  - SLAM, BLAST
  - (many, many non-VC approaches)
• Automatic theorem proving
  - Simplify, SVC, CVC, Touchstone
• Constraint Logic Programming
  - [Jaffar and Lassez, POPL’87], SICStus Prolog, ...
• CLP for model checking
  - [Delzanno and Podelski]
• Interactive theorem proving
  - PVS, HOL, Isabelle, ACL2
Summary

- Deep connection between
  - correctness of imperative programs
    - with pointers, heap allocation, aliasing, ...
  - satisfiability of CLP queries

- Verifun Checker
  - interprocedural extended static checker
  - reduced annotation burden
  - statically check assertions, API usage rules, ...
  - interprocedural ECs are constraint logic programs
Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries

Cormac Flanagan
University of California, Santa Cruz