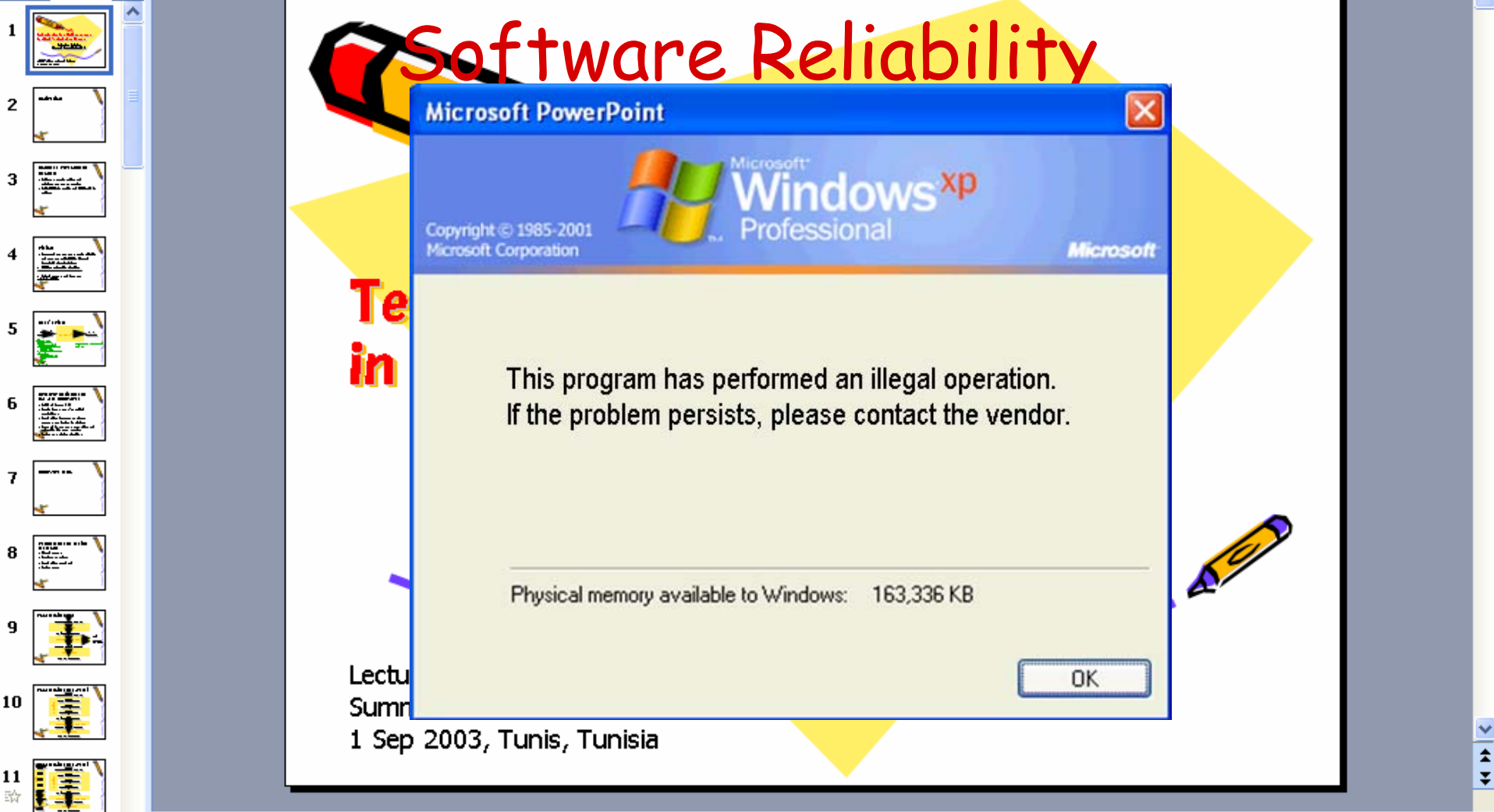



Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries

Cormac Flanagan
University of California, Santa Cruz



1 

Software Reliability

Te in

Lectu
Summ

1 Sep 2003, Tunis, Tunisia

Microsoft PowerPoint



Microsoft
Windows^{XP}
Professional

Copyright © 1985-2001
Microsoft Corporation

Microsoft

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

Physical memory available to Windows: 163,336 KB

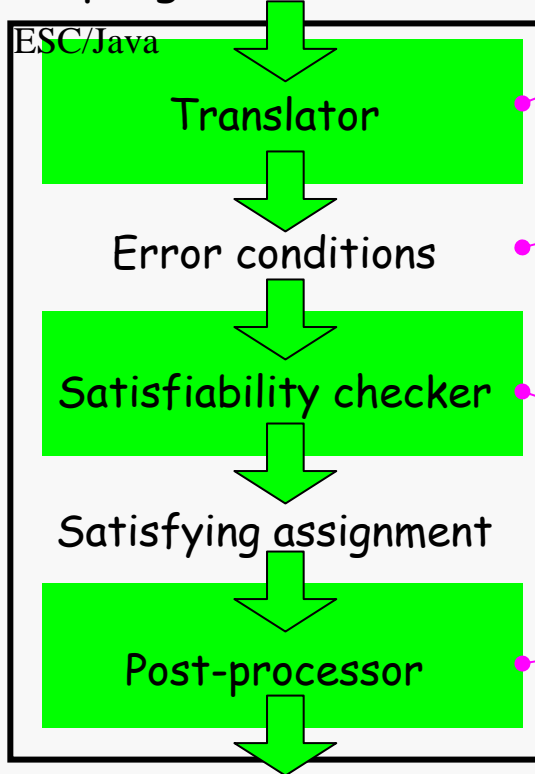
OK

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

ESC/Java Architecture

Java program + Annotations



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.

Index out of bounds on line 218

Method does not preserve object invariant on line 223

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;
  }

  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() );
    }
  }
}
```

Warning: invariant possibly not established

Warning: possible division by zero

Warning: precondition possibly not established

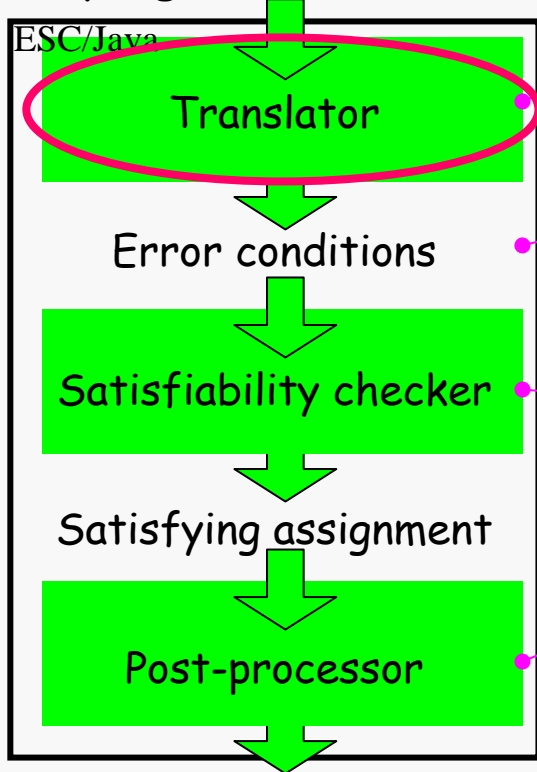


ESC/Java Experience

- 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?**

ESC/Java Architecture

Java program + 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.

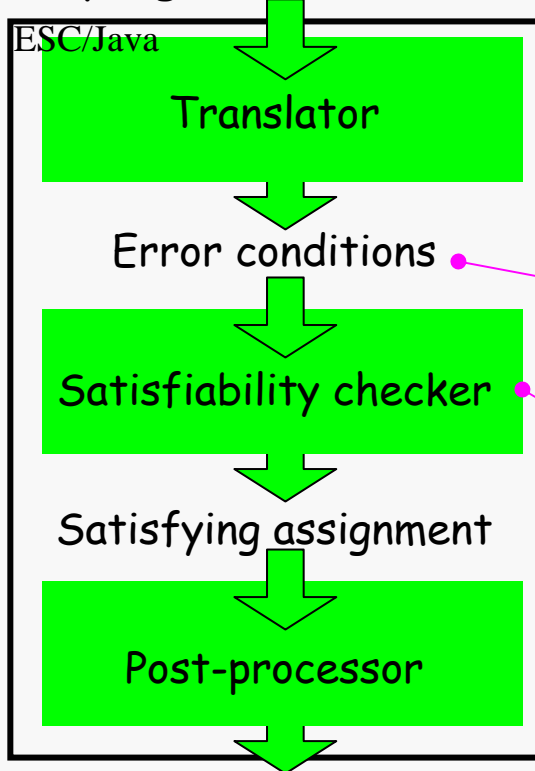
Satisfying assignments are turned into precise warning messages.

Index out of bounds on line 218

Method does not preserve object invariant on line 223

Generating Error Conditions

Java program + Annotations



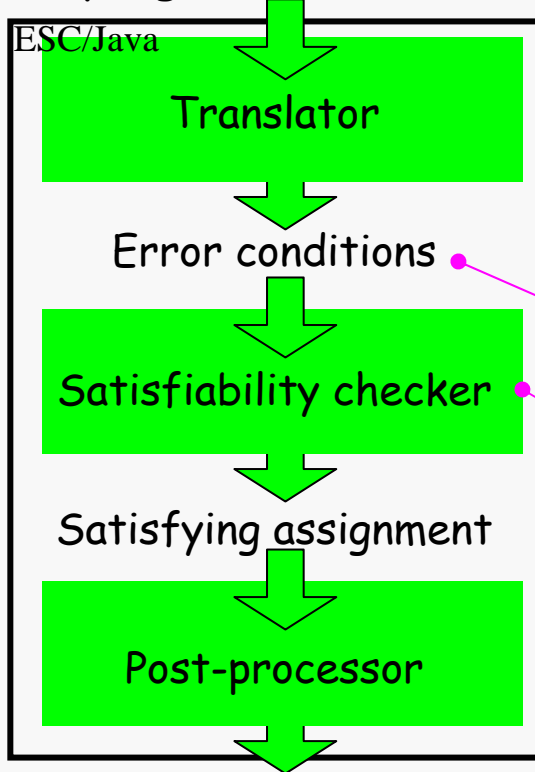
`if (x < 0) { x := -x; }
/*@ assert x >= 0;`

$(x < 0 \wedge x' = -x \wedge \neg(x' \geq 0))$
 $\vee (\neg(x < 0) \wedge \neg(x \geq 0))$

Unsatisfiable, no error

Generating Error Conditions 2

Java program + Annotations

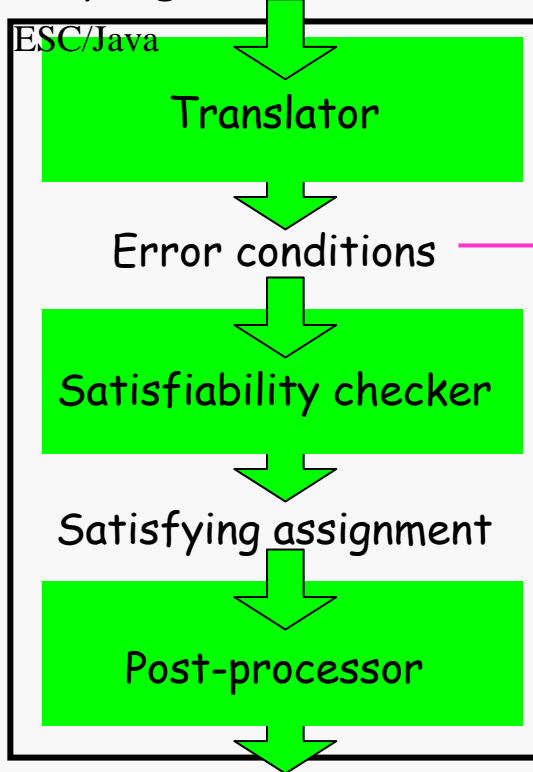


`p := q;`
`p.f := 3;`
`//@ assert q.f != 0;`

$p = q$
 $\wedge f' = \text{store}(f, p, 3)$
 $\wedge \text{select}(f', q) = 0$
Unsatisfiable, no error

ESC/Java Error Condition Logic

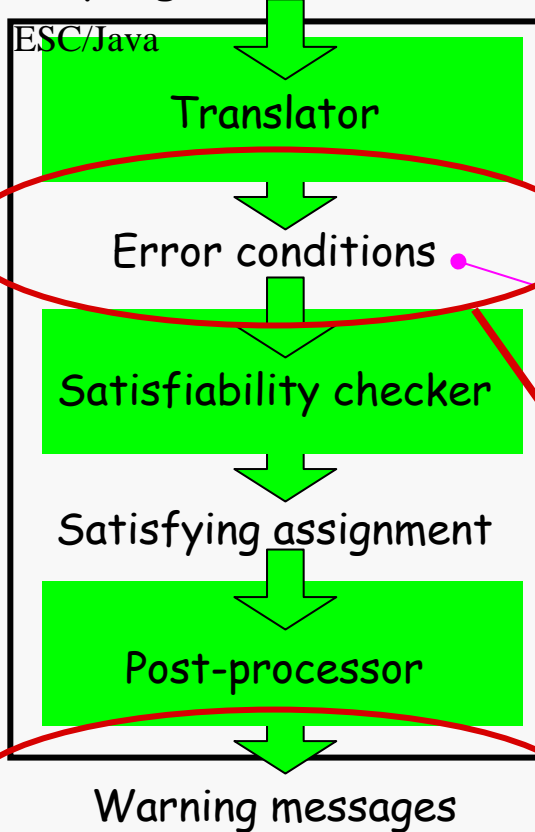
Java program + Annotations



- terms $t ::= x \mid f(\underline{t})$
- constraints $c ::= p(\underline{t})$
- formulae $e ::= c \mid \neg c \mid e \wedge e \mid e \vee e \mid \exists y. e$
- theories: equality, linear arithmetic, select+store
- some heuristics for universal quantification
- logic cannot express iteration or recursion

Error Conditions for Procedures

Java program + Annotations



call p(x);
assert x>0

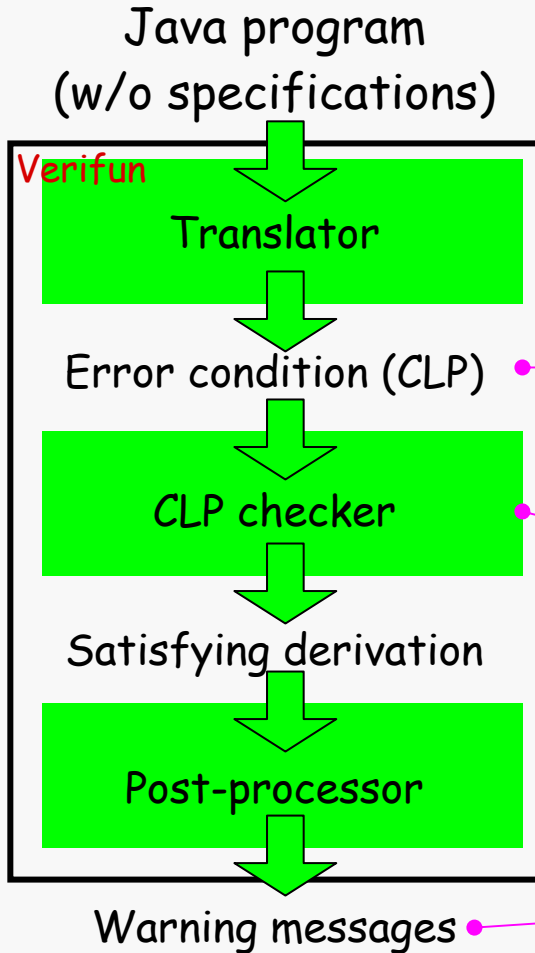
```
//@ requires pre(x)  
//@ ensures post(x,x')  
void p(x) { ... }
```

$\neg \text{pre}(x)$
 $\vee \exists x'. (\text{post}(x,x') \wedge \neg(x'>0))$

- 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

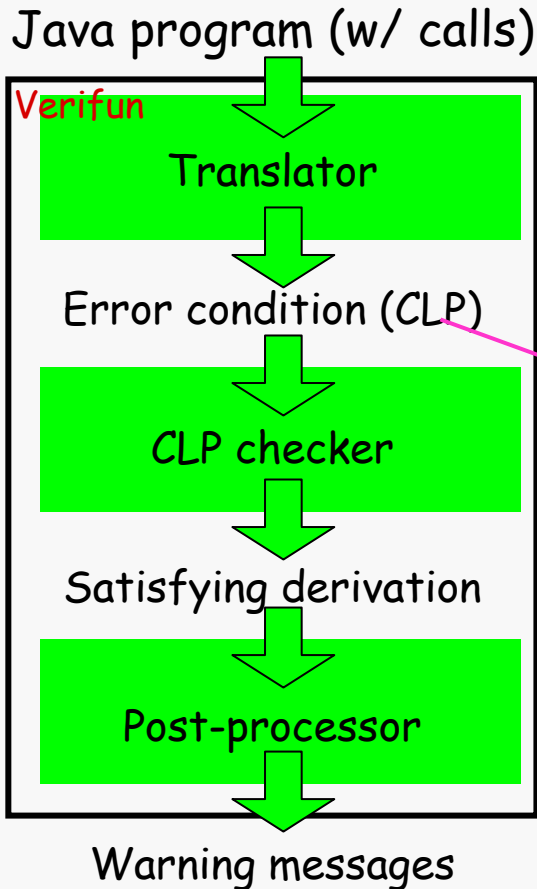
- can express relation definitions and calls

Satisfiability checker for CLP

The error message includes a complete execution trace

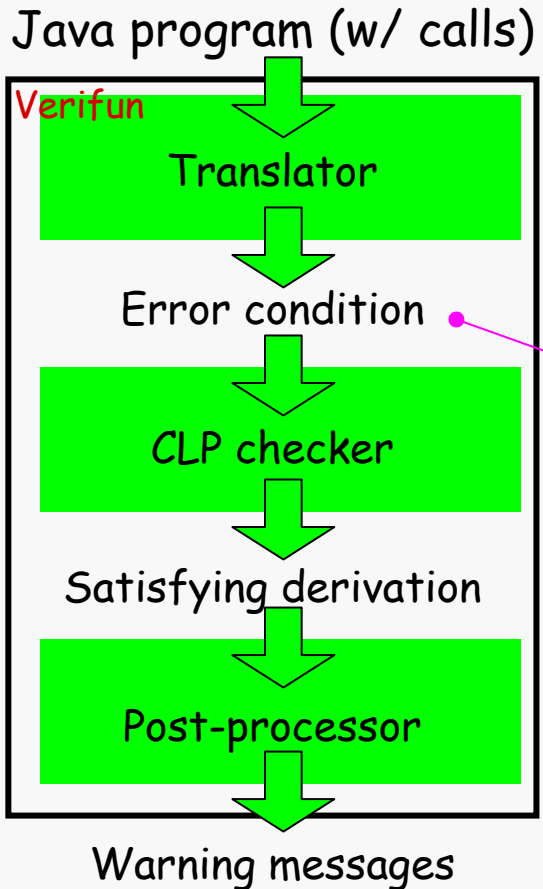
- no warnings about missing specifications

Verifun Architecture



- terms $t ::= x \mid f(t)$
- constraints $c ::= p(t)$
- formulas $e ::= c \mid \neg c \mid e \wedge e \mid e \vee e \mid \exists y. e \mid r(t)$
- 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



call p(x);
assert x>0

```
void p(x) { S }
```

$E_p(x)$
 $\vee \exists \underline{x}'. (Tp(\underline{x}, \underline{x}') \wedge \neg(x' > 0))$

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

Example: Factorial

```
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);
}
```

$T_{\text{fact}}(i, r) :-$
 $(i = 0 \wedge r = 1)$
 $\vee (i \neq 0 \wedge T_{\text{fact}}(i-1, t)$
 $\wedge t > 0 \wedge r = i * t)$

$E_{\text{fact}}(i) :-$
 $i \neq 0$
 $\wedge (E_{\text{fact}}(i-1)$
 $\vee (T_{\text{fact}}(i-1, t) \wedge t \leq 0))$

$E_{\text{main}}() :- E_{\text{fact}}(j)$

- $T_{\text{fact}}(i, r)$ relates pre and post states of executions of `fact`
- $E_{\text{fact}}(i)$ describes pre-states from which `fact` may go wrong
- CLP has least fixpoint semantics
- CLP Query: Is $E_{\text{main}}()$ 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

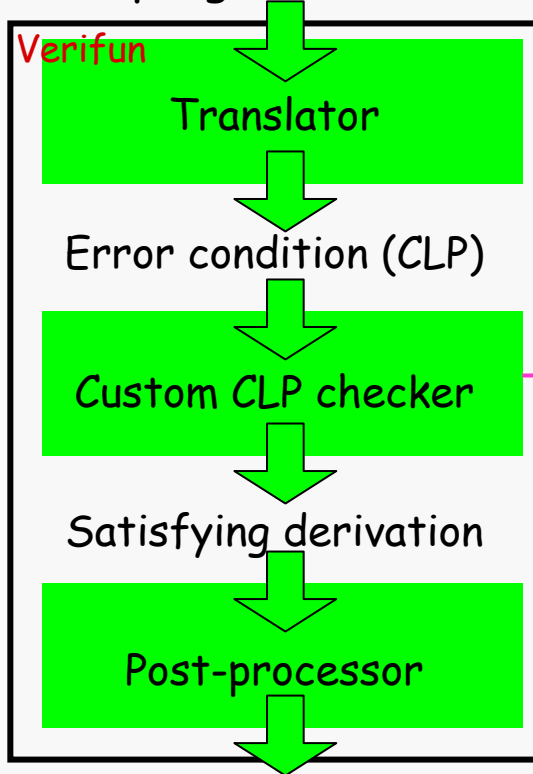
```
class Rational {  
  
    int num, den;  
  
    Rational(int n, int d) {  
        num = n;  
        den = d;  
    }  
  
    int trunc() {  
        return num/den;  
    }  
  
    public static void main(String[] a) {  
        int n = readInt(), d = readInt();  
        if( d == 0 ) return;  
        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

Java program (w/ calls)



- 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:

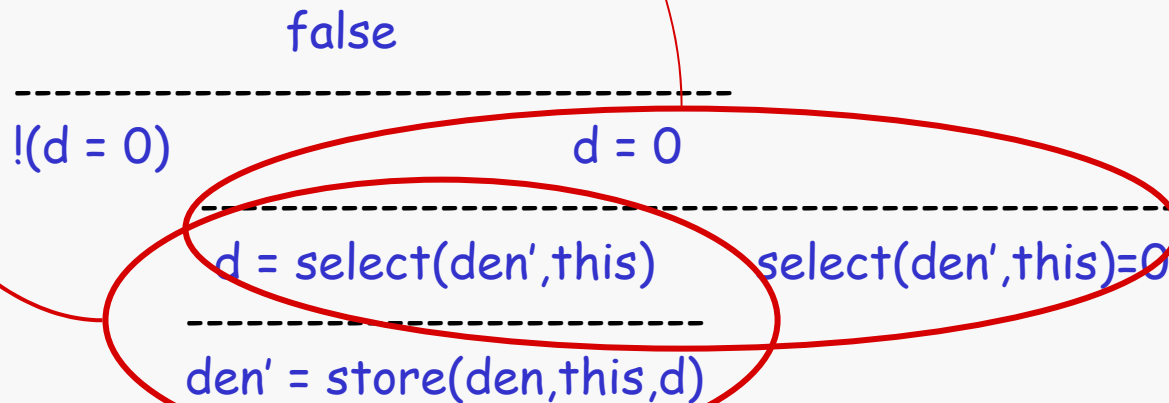
$\wedge \{d \neq 0\}$
 $\wedge \vee \{num' = store(num, this, n)\}$
 $\vee \{num' = store(num, this, 0)\}$
 $\wedge \{den' = store(den, this, d)\}$
 $\wedge \{select(den', this) = 0\}$
 $\wedge \{d = select(den', this)\} \wedge \{select(den', this) = 0\} \Rightarrow \{d = 0\}$
 $\wedge \{den' = store(den, this, d)\} \Rightarrow \{d = select(den', this)\}$

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

Unsatisfiable!

Truth assignment TA:

$!(d = 0)$
 $num' = store(num, this, n)$
 $den' = store(den, this, d)$
 $select(den', this) = 0$



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 $r(\underline{x}) :- e$
- to *propositional* relation definition $R(\underline{B}) :- E$
 - \underline{B} is a list of boolean variables
 - E is a propositional formula
 - satisfiability is decidable

Abstraction Refinement for CLP - 1

Program:

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

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

Error Condition Q:

$Tinc(x,x') :-$
 $\wedge y=x$
 $\wedge \vee x'=0$
 $\vee x'=y+1$

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

Is $Emain()$ satisfiable?

Abstract EC Q^a:

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

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

Is $EMAIN()$ sat?

Abstraction Refinement for CLP - 2

Abstract EC Q^a:

TINC() :-

$$\begin{aligned} &\wedge \{y=x\} \\ &\wedge \vee \{x'=0\} \\ &\quad \vee \{x'=y+1\} \end{aligned}$$

EMAIN() :-

$$\begin{aligned} &\wedge \{z=0\} \\ &\wedge \text{TINC()} \\ &\wedge \neg\{z'=1\} \end{aligned}$$

Is **EMAIN()** sat?

Abstract Trace T^a:

TINC() :-

$$\begin{aligned} &\wedge \{y=x\} \\ &\wedge \{x'=y+1\} \end{aligned}$$

EMAIN() :-

$$\begin{aligned} &\wedge \{z=0\} \\ &\wedge \text{TINC()} \\ &\wedge \neg\{z'=1\} \end{aligned}$$

EMAIN() is sat.

Concrete Trace T:

Tinc(x,x') :-

$$\begin{aligned} &\wedge y=x \\ &\wedge x'=y+1 \end{aligned}$$

Emain() :-

$$\begin{aligned} &\wedge z=0 \\ &\wedge \text{Tinc}(z,z') \\ &\wedge \neg(z'=1) \end{aligned}$$

EMAIN() is sat.

Abstraction Refinement for CLP - 3

Concrete Trace T:

Conjunction of Atoms:

Tinc(x,x') :-

$\wedge y=x$
 $\wedge x'=y+1$

$\wedge y=x$
 $\wedge x'=y+1$

Emain() :-

$\wedge z=0$
 $\wedge \text{Tinc}(z,z')$
 $\wedge \neg(z'=1)$

$\wedge z=0$
 $\wedge z=x \wedge z'=x'$
 $\wedge \neg(z'=1)$

Abstraction Refinement for CLP - 4

Conjunction of Atoms:

$$\wedge y=x$$

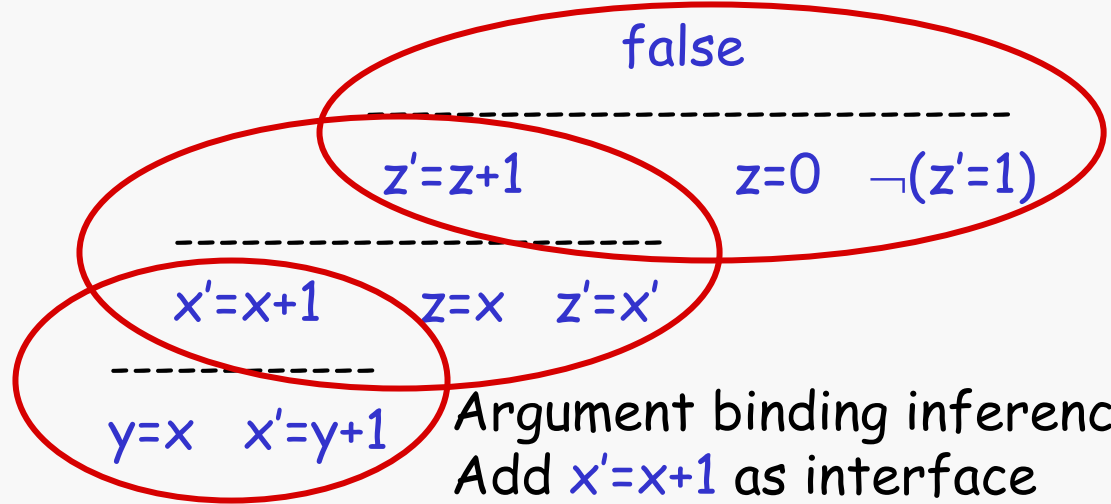
$$\wedge x'=y+1$$

$$\wedge z=0$$

$$\wedge z=x \wedge z'=x'$$

$$\wedge \neg(z'=1)$$

Explicated clause in `Emain()`
 $z'=z+1 \wedge z=0 \wedge \neg(z'=1) \Rightarrow \text{false}$



Explicated clause in `Tinc()`
 $y=x \wedge x'=y+1 \Rightarrow x'=x+1$

Argument binding inference
 Add $x'=x+1$ as interface
 predicate for `Tinc()`

Abstraction Refinement for CLP - 5

Error Condition Q:

Tinc(x,x') :-

$\wedge y=x$
 $\wedge \vee x'=0$
 $\vee x'=y+1$

Emain() :-

$\wedge z=0$
 $\wedge \text{Tinc}(z,z')$
 $\wedge \neg(z'=1)$

Abstract EC Q^a:

TINC({x'=x+1}) :-

$\wedge \{y=x\}$
 $\wedge \vee \{x'=0\}$
 $\vee \{x'=y+1\}$
 $\wedge (\{y=x\} \wedge \{x'=y+1\} \Rightarrow \{x'=x+1\})$

EMAIN() :-

$\wedge \{z=0\}$
 $\wedge \text{TINC}(\{z'=z+1\})$
 $\wedge \neg\{z'=1\}$
 $\wedge \neg(\{z'=z+1\} \wedge \{z=0\} \wedge \neg\{z'=1\})$

Abstraction Refinement for CLP - 6

Abstract EC Qa:

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

EMAIN() :-
 $\wedge \{z=0\}$
 \wedge TINC($\{z'=z+1\}$)
 $\wedge \neg\{z'=1\}$
 $\wedge \neg(\{z'=z+1\} \wedge \{z=0\} \wedge \neg\{z'=1\})$

Abstract Trace T^a:

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

EMAIN() :-
 $\wedge \{z=0\}$
 \wedge TINC($\{z'=z+1\}$)
 $\wedge \neg\{z'=1\}$
 $\wedge \neg\{z'=z+1\}$

Abstraction Refinement for CLP - 7

Abstract Trace T^a :

TINC($\{x'=x+1\}$) :-

$\wedge \{y=x\}$
 $\wedge \{x'=0\}$
 $\wedge \neg\{x'=y+1\}$
 $\wedge \neg\{x'=x+1\}$

EMAIN() :-

$\wedge \{z=0\}$
 $\wedge \text{TINC}(\{z'=z+1\})$
 $\wedge \neg\{z'=1\}$
 $\wedge \neg\{z'=z+1\}$

Concrete Trace T :

TINC(x,x) :-

$\wedge y=x$
 $\wedge x'=0$
 $\wedge \neg(x'=y+1)$
 $\wedge \neg(x'=x+1)$

EMAIN() :-

$\wedge z=0$
 $\wedge \text{TINC}(z,z')$
 $\wedge \neg(z'=1)$
 $\wedge \neg(z'=z+1)$

Conjunction of Constraints

$\wedge y=x$
 $\wedge x'=0$
 $\wedge \neg(x'=y+1)$
 $\wedge \neg(x'=x+1)$

$\wedge z=0$
 $\wedge z=x \wedge z'=x'$
 $\wedge \neg(z'=1)$
 $\wedge \neg(z'=z+1)$

Abstraction Refinement for CLP - 7

Conjunction of Constraints:

$\wedge y=x$
 $\wedge x'=0$
 $\wedge \neg(x'=y+1)$
 $\wedge \neg(x'=x+1)$

$\wedge z=0$
 $\wedge z=x \wedge z'=x'$
 $\wedge \neg(z'=1)$
 $\wedge \neg(z'=z+1)$

decision
procedures say
this conjunction
of constraints is
consistent,
so this is a real
error trace

Program Trace:

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

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)
 \wedge select(den',r)=0
 $\Rightarrow d = 0$

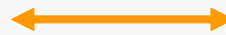
Properties of Verifun CLP Checker

- 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

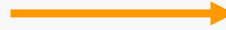

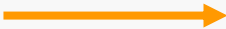
Unit of Specification

- 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



Constraint Logic Programming

- Program correctness  • CLP satisfiability
- Bounded software model checking  • Efficient implementations
 - Sicstus Prolog
 - depth-first
- Explicating theorem proving  • CLP implementation technique
 - Avoids considering all paths
 - Verifun CLP satisfiability checker
- Predicate abstraction & predicate inference
 - SLAM, BLAST

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