Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries

Cormac Flanagan
University of California, Santa Cruz

Microsoft PowerPoint 2003 - [Technologies for finding errors in object-oriented software, Lecture 0.ppt]

File Edit View Insert Format Tools Slide Show Window Help

Software Reliability

Te
in

Lectu
Summ
1 Sep 2003, Tunis, Tunisia

Click to add notes

Slide 1 of 47 Crayons

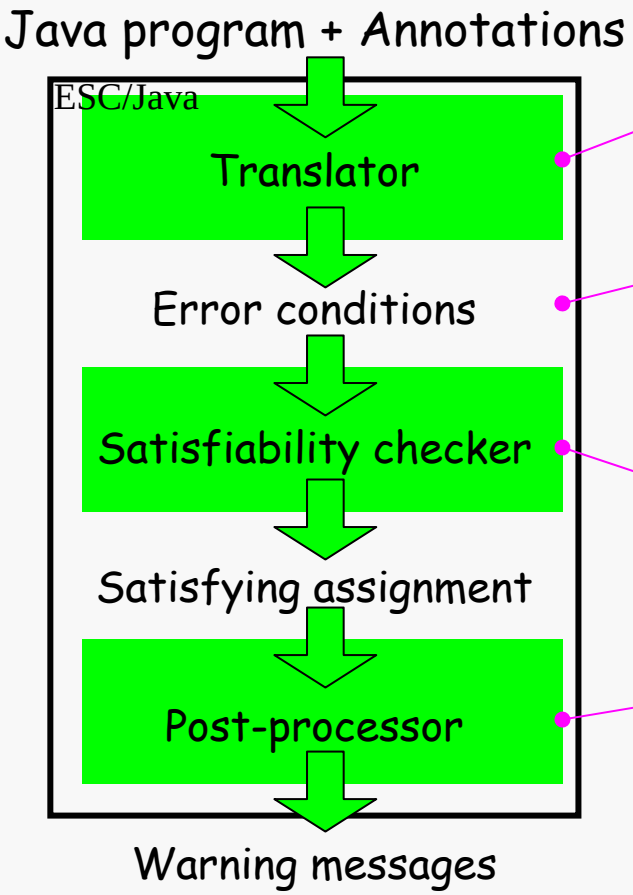
start Technologies for findi... 4:35

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



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

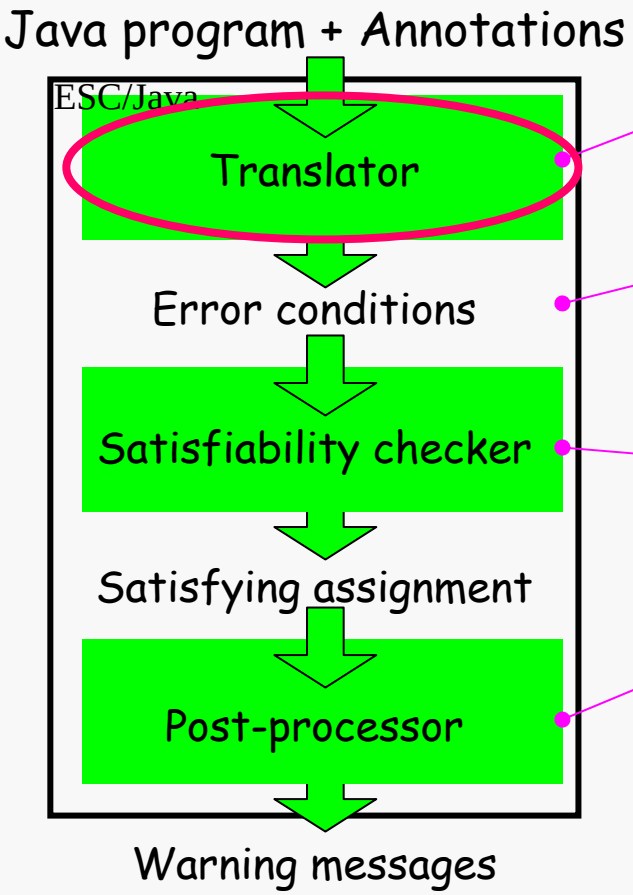
Warning: possible division by zero

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



The translator "understands" the semantics of Java.

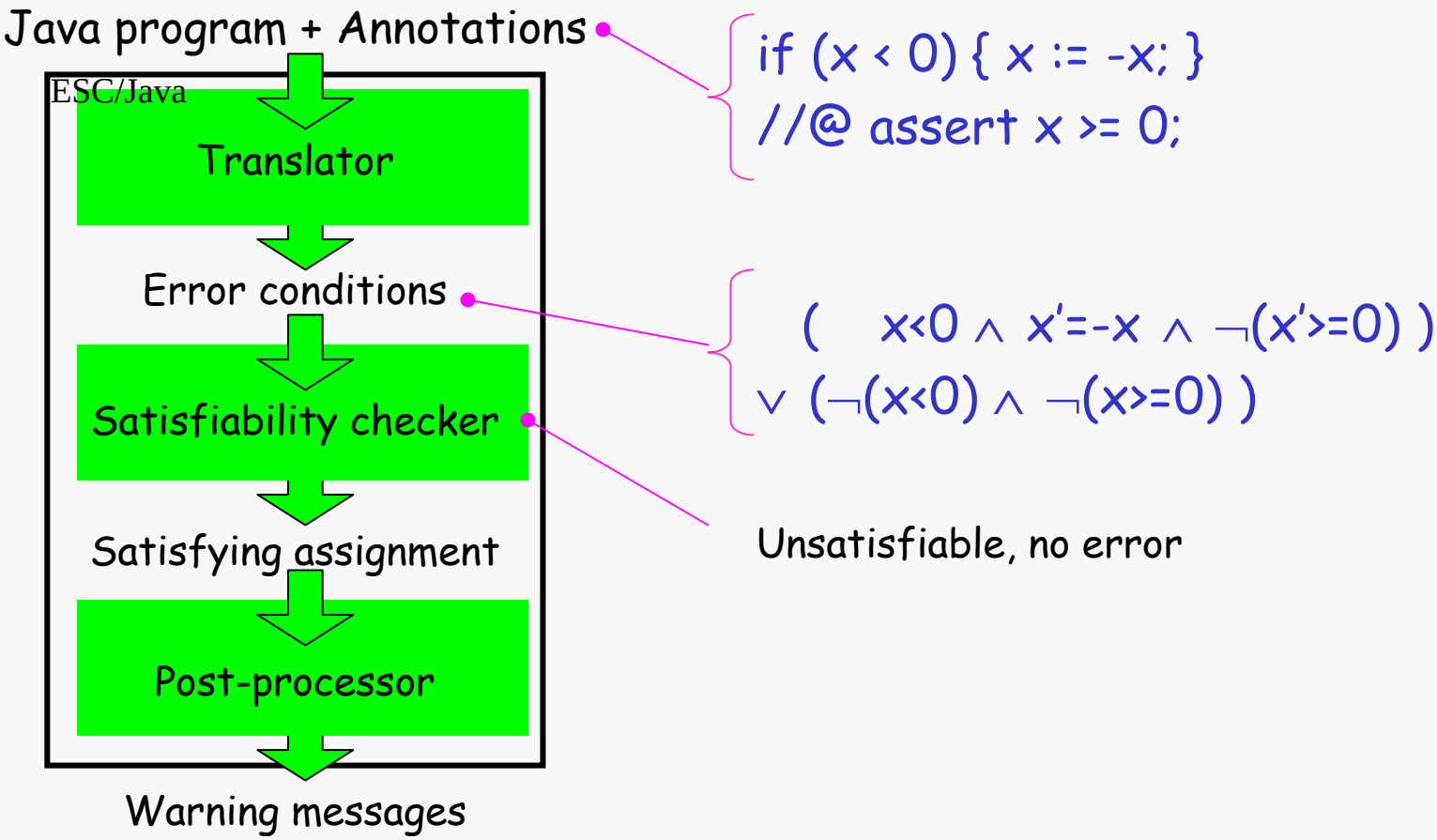
An *error condition* is a logical formula that, ideally, is satisfiable if and only if the program contains an error.

The satisfiability checker is invisible to users.

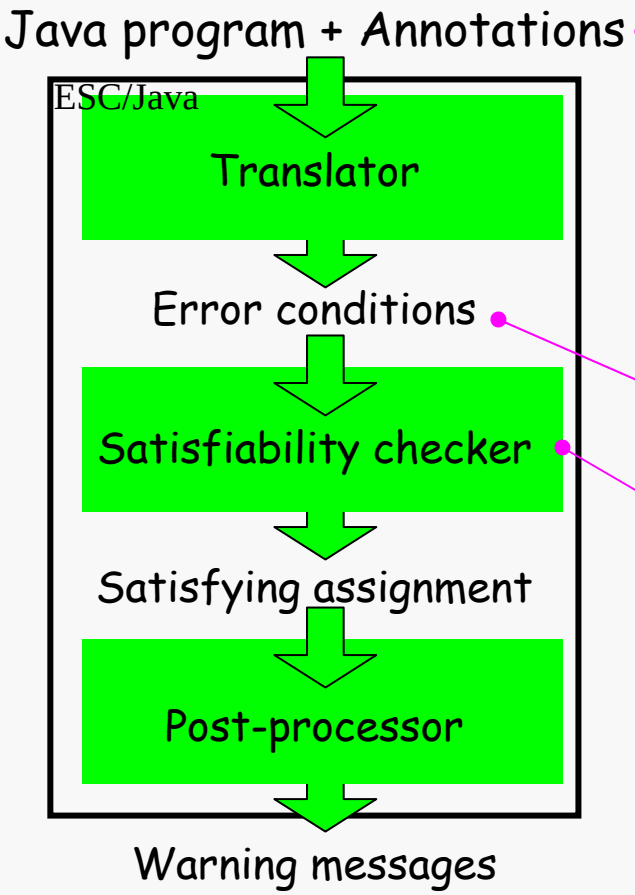
Satisfying assignments are turned into precise warning messages.

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

Generating Error Conditions



Generating Error Conditions 2

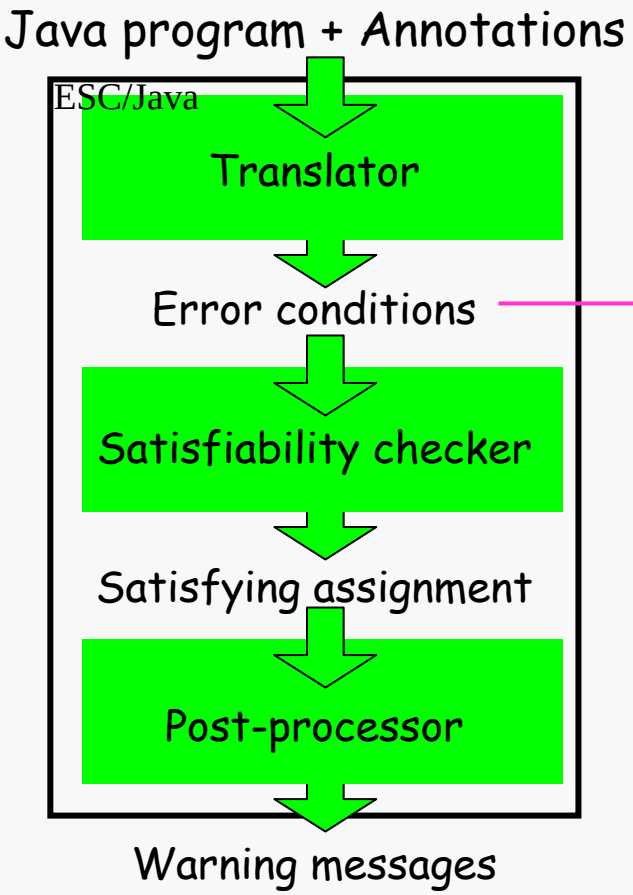



```
p := q;
p.f := 3;
//@ assert q.f != 0;
```

$$
\begin{aligned}
& p = q \\
\wedge\ & f' = \mathrm{store}(f, p, 3) \\
\wedge\ & \mathrm{select}(f', q) = 0
\end{aligned}
$$

Unsatisfiable, no error

ESC/Java Error Condition Logic

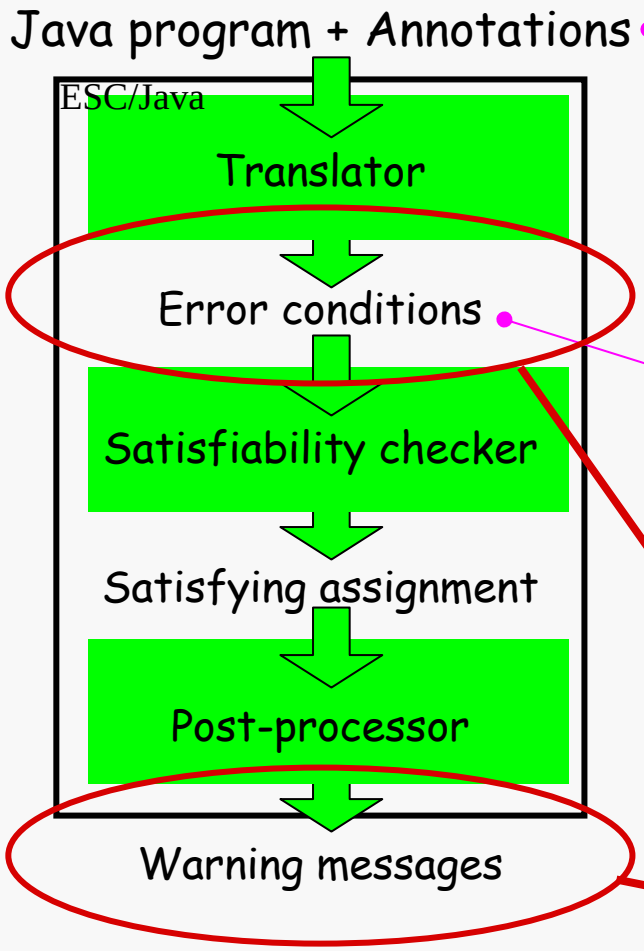


- terms $t ::= x \mid f(\underline{t})$
- constraints $c ::= p(\underline{t})$
- formulae $e ::= c \mid \neg c \mid e \wedge e \mid e \vee e \mid \exists y.\, e$

- theories: equality, linear arithmetic, select+store
- some heuristics for universal quantification

- logic cannot express iteration or recursion

Error Conditions for Procedures



call p(x);
assert x>0

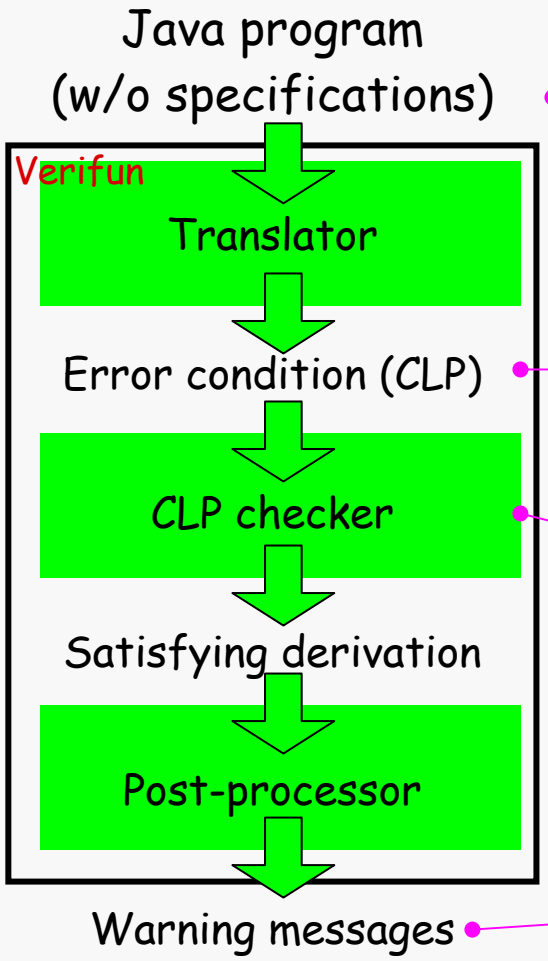
$\neg pre(x)$
$\vee \exists x'. (post(x,x') \wedge \neg(x'>0))$

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

- Programmer must write procedure specs
- so procedure calls can be translatec 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

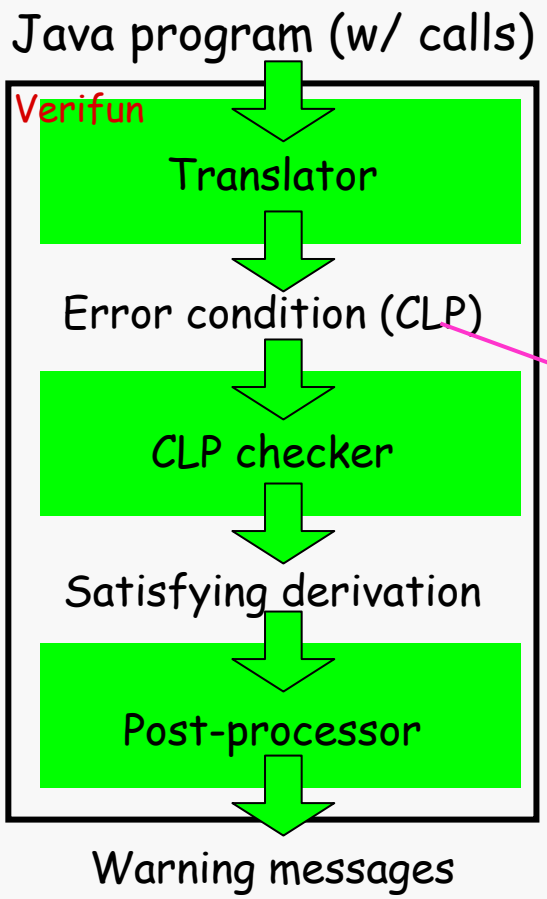
- can express relation definitions and calls

Satisfiability checker for CLP

The error message includes a complete execution trace

- no warnings about missing specifications

Verifun Architecture

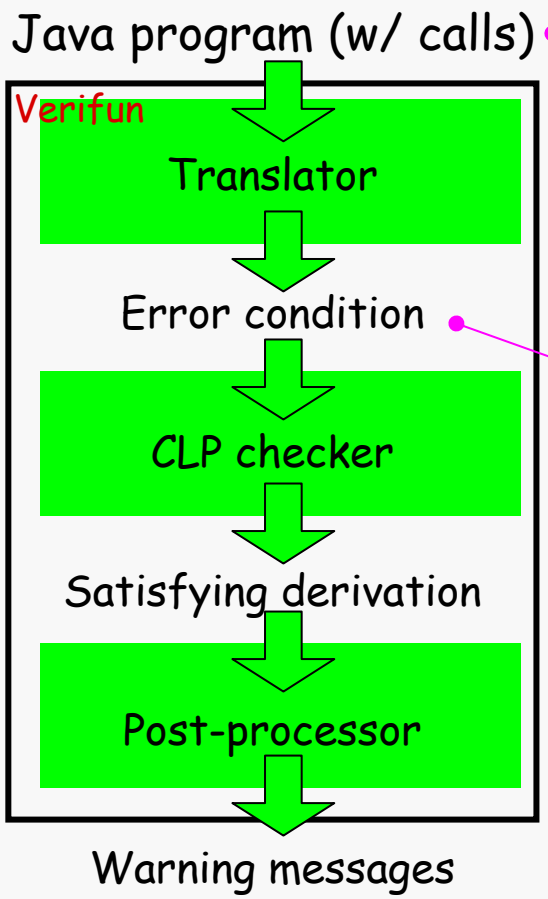


- terms $t ::= x \mid f(\underline{t})$
- constraints $c ::= p(\underline{t})$
- formulas $e ::= c \mid \neg c \mid e \wedge e \mid e \vee e \mid \exists \underline{y}.\, e \mid r(\underline{t})$
- user-defined relation symbols r
- definitions $d ::= r(\underline{x}) :- e$

- Query: Given $\underline{d}$, is e satisfiable?

- Constraint Logic Programming
  - [Jaffar and Lassez, POPL'87]
  - Efficient implementations! ☺

Error Conditions for Procedures



call p(x);
assert x>0

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

void p(x) { S }

- 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

```
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);
}
```

```
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		Constraint Logic Programming
• Program correctness	→	• CLP satisfiability
• Bounded software model checking	←	• 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

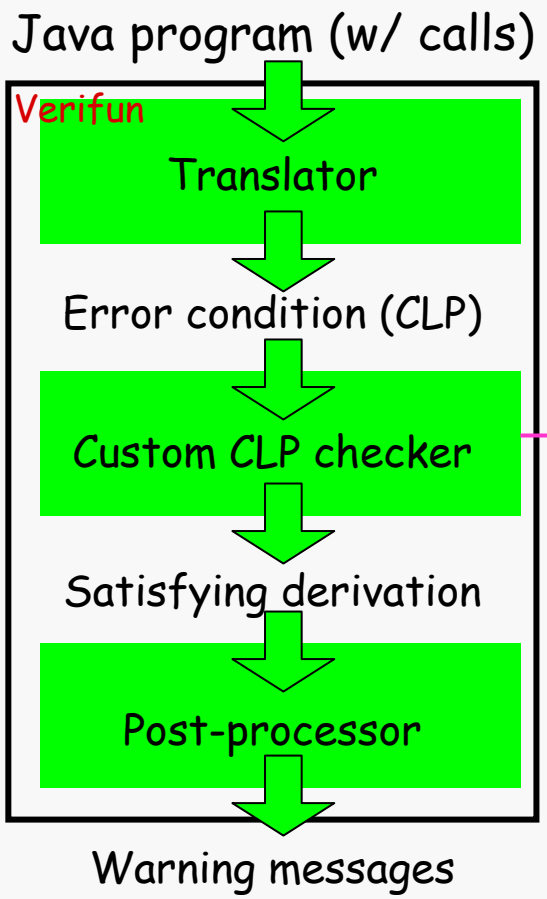
```
class Rational {

  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

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

<u>Query Q^b:</u>

$\wedge$!{d = 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} => {d=0}

$\wedge$ {den′=store(den,this,d)} => {d=select(den′,this)}

Unsatisfiable!

<u>Truth assignment TA:</u>

!(d = 0)

num′ = store(num,this,n)

den′ = store(den,this,d)

select(den′,this) = 0

false

!(d = 0) d = 0

---

d = select(den′,this) select(den′,this)=0

---

den′ = store(den,this,d)

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:

$$\begin{aligned} &Tinc(x,x') :- \\ &\quad \wedge\ y=x \\ &\quad \wedge\ \vee\ x'=0 \\ &\quad\quad \vee\ x'=y+1 \end{aligned}$$

$$\begin{aligned} &Emain() :- \\ &\quad \wedge\ z=0 \\ &\quad \wedge\ Tinc(z,z') \\ &\quad \wedge\ \neg(z'=1) \end{aligned}$$

Is Emain() satisfiable?

Abstract EC Q^a:

$$\begin{aligned} &TINC() :- \\ &\quad \wedge\ \{y=x\} \\ &\quad \wedge\ \vee\ \{x'=0\} \\ &\quad\quad \vee\ \{x'=y+1\} \end{aligned}$$

$$\begin{aligned} &EMAIN() :- \\ &\quad \wedge\ \{z=0\} \\ &\quad \wedge\ TINC() \\ &\quad \wedge\ \neg\{z'=1\} \end{aligned}$$

Is EMAIN() sat?

Abstraction Refinement for CLP - 2

<u>Abstract EC Q^a</u>:

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?

<u>Abstract Trace T^a</u>:

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

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

EMAIN() is sat.

<u>Concrete Trace T</u>:

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

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

EMAIN() is sat.

Abstraction Refinement for CLP - 3

Concrete Trace T:	Conjunction of Atoms:
Tinc(x,x') :-	
$\wedge$ y=x	$\wedge$ y=x
$\wedge$ x'=y+1	$\wedge$ x'=y+1
Emain() :-	
$\wedge$ z=0	$\wedge$ z=0
$\wedge$ Tinc(z,z')	$\wedge$ z=x $\wedge$ z'=x'
$\wedge$ $\neg$(z'=1)	$\wedge$ $\neg$(z'=1)

Abstraction Refinement for CLP - 4

<u>Conjunction of Atoms</u>:

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

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

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

false

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

$x'=x+1$ $z=x$ $z'=x'$

$y=x$ $x'=y+1$

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

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

Abstraction Refinement for CLP - 5

<u>Error Condition Q</u>:

$$
\begin{aligned}
&\mathrm{Tinc}(x,x') \;:\!-\\
&\qquad \wedge\; y=x\\
&\qquad \wedge\; \vee\; x'=0\\
&\qquad\quad \vee\; x'=y+1
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Emain}() \;:\!-\\
&\qquad \wedge\; z=0\\
&\qquad \wedge\; \mathrm{Tinc}(z,z')\\
&\qquad \wedge\; \neg(z'=1)
\end{aligned}
$$

<u>Abstract EC Q^a</u>:

$$
\begin{aligned}
&\mathrm{TINC}(\{x'=x+1\}) \;:\!-\\
&\qquad \wedge\; \{y=x\}\\
&\qquad \wedge\; \vee\; \{x'=0\}\\
&\qquad\quad \vee\; \{x'=y+1\}\\
&\qquad \wedge\; (\{y=x\} \wedge \{x'=y+1\} \Rightarrow \{x'=x+1\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{EMAIN}() \;:\!-\\
&\qquad \wedge\; \{z=0\}\\
&\qquad \wedge\; \mathrm{TINC}(\{z'=z+1\})\\
&\qquad \wedge\; \neg\{z'=1\}\\
&\qquad \wedge\; \neg(\{z'=z+1\} \wedge \{z=0\} \wedge \neg\{z'=1\})
\end{aligned}
$$

Abstraction Refinement for CLP - 6

<u>Abstract EC Qa</u>:

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

<u>Abstract Trace T^a</u>:

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:	Concrete Trace T:	Conjunction of Constraints
TINC({x'=x+1}) :-	TINC(x,x) :-	
$\wedge$ {y=x}	$\wedge$ y=x	$\wedge$ y=x
$\wedge$ {x'=0}	$\wedge$ x'=0	$\wedge$ x'=0
$\wedge \neg$\{x'=y+1\}	$\wedge \neg$(x'=y+1)	$\wedge \neg$(x'=y+1)
$\wedge \neg$\{x'=x+1\}	$\wedge \neg$(x'=x+1)	$\wedge \neg$(x'=x+1)
EMAIN() :-	EMAIN() :-	
$\wedge$ {z=0}	$\wedge$ z=0	$\wedge$ z=0
$\wedge$ TINC({z'=z+1})	$\wedge$ TINC(z,z')	$\wedge$ z=x $\wedge$ z'=x'
$\wedge \neg$\{z'=1\}	$\wedge \neg$(z'=1)	$\wedge \neg$(z'=1)
$\wedge \neg$\{z'=z+1\}	$\wedge \neg$(z'=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
=> 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 →
- Bounded software model checking ←

- CLP satisfiability
- Efficient implementations
  - Sicstus Prolog
  - depth-first

- Explicating theorem proving
- Predicate abstraction & predicate inference
  - SLAM, BLAST

→

- 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