Verifying Programs via Iterated Specialization

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1]University of Chieti-Pescara 'G. d'Annunzio'
[2]University of Rome 'Tor Vergata'
[3]CNR - Istituto di Analisi dei Sistemi ed Informatica, Rome

PEPM 2013
Rome, Italy, 21 January 2013

Summary

- Software Model Checking of imperative programs...
  - Safety properties of C programs
- ... by iterated specialization of Constraint Logic Programs
  - First specialization:
    removal of the interpreter
  - Subsequent specializations:
    one or more propagations of constraints
    of the initial or error configurations
- Experimental results

Program Safety

P:
```
void main(){
  int x;
  int y;
  int n;
  while(x < n) {
    x=x+1;
    y=x+y;
  }
}
```

$\varphi_{init}(\mathtt{x},\mathtt{y},\mathtt{n}) \equiv \mathtt{x} = 0 \wedge \mathtt{y} = 0$

$\varphi_{error}(\mathtt{x},\mathtt{y},\mathtt{n}) \equiv \mathtt{x} > \mathtt{y}$

A program P is safe w.r.t. φ_{init} and φ_{error} if
from any configuration satisfying φ_{init}
no configuration satisfying φ_{error} can be reached.
Otherwise, program P is unsafe.

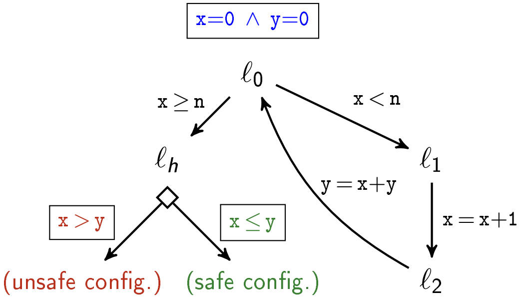
Safety Verification as a Reachability Problem

- Program execution as a transition relation.

program P:

```
     void main(){
       int x;
       int y;
       int n;
ℓ0 :    while(x < n) {
ℓ1 :       x = x+1;
ℓ2 :       y = x+y;
         }
ℓh : }
```

execution of P:



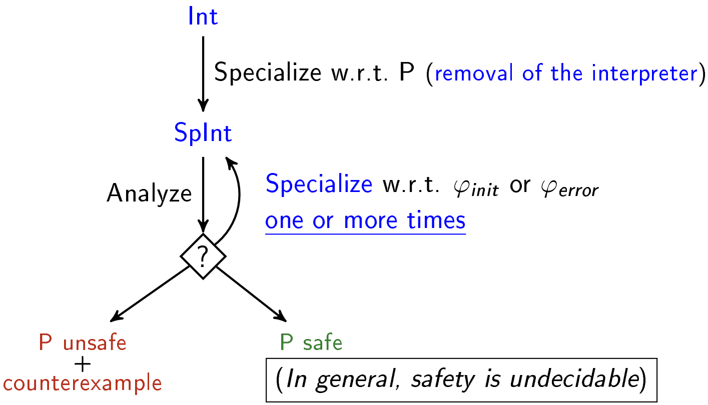
Related Work

- Static analysis and model checking
  - Cousot and Cousot.
    Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. [POPL'78]
    ...
  - Saïdi. *Model checking guided abstraction and analysis.* [SAS'00]
- Constraint-based verification
  - Podelski and Rybalchenko.
    ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. [PADL'07]
  - Jaffar, Navas, and Santosa.
    TRACER: A Symbolic Execution Tool for Verification [CAV'12]
  - Grebenshchikov, Gupta, Lopes, Popeea, and Rybalchenko.
    HSF(C): A Software Verifier based on Horn Clauses. [TACAS'12]
- Specialization-based verification
  - Peralta, Gallagher, and Saglam.
    Analysis of Imperative Programs through Analysis of Constraint Logic Programs. [SAS'98]

Verification Framework

We use a Constraint Logic Program (CLP) program for encoding:

- the program P to be verified (written in the language C)
- the interpreter Int (i.e., the semantics of the language C)
- the configurations φ_{init} or φ_{error}



Encoding of P

program P:

```
      void main(){
        int x;
        int y;
        int n;
ℓ0 :    while(x < n) {
ℓ1 :      x=x+1;
ℓ2 :      y=x+y;
ℓ3 :    }
ℓh :  }
```

encoding of program P:

$\mathtt{at}(\ell_0, \mathtt{ite}(\mathtt{less}(\mathtt{int}(\mathtt{x}), \mathtt{int}(\mathtt{n})), \ell_1, \ell_h)).$

$\mathtt{at}(\ell_1, \mathtt{asgn}(\mathtt{int}(\mathtt{x}), \mathtt{plus}(\mathtt{int}(\mathtt{x}), \mathtt{int}(1)))).$

$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{int}(\mathtt{y}), \mathtt{plus}(\mathtt{int}(\mathtt{x}), \mathtt{int}(\mathtt{y})))).$

$\mathtt{at}(\ell_3, \mathtt{goto}(\ell_0)).$

$\mathtt{at}(\ell_h, \mathtt{halt}).$

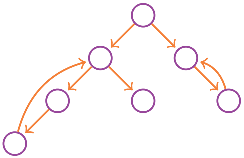
- a set of configurations: $\mathtt{cf}(\mathtt{C},\mathtt{S})$ $(\bigcirc)$

  Each configuration is made out of :
  - a command $\mathtt{C}$
  - a state $\mathtt{S}$: a list of [variable, value] pairs

    for instance: $[[\mathtt{int}(\mathtt{x}),\mathtt{x1}],\ [\mathtt{int}(\mathtt{y}),\mathtt{y1}]]$

- a transition relation: $\mathtt{tr}(\mathtt{cf}(\mathtt{C},\mathtt{S}),\ \mathtt{cf}(\mathtt{C1},\mathtt{S1}))$ $(\rightarrow)$

  (i.e., operational semantics)



Encoding of the Interpreter of C (2)

`Id = Expr;`	`tr(cf(L,asgn(Id,Expr),S),cf(C,S1)) :- aeval(Expr,S,V), update(Id,V,S,S1), nextlab(L,C).`
`if (Expr) { goto L1; } else goto L2; }`	`tr(cf(ite(Expr,L1,L2),S),cf(C,S)) :- beval(Expr,S), at(L1,C).` `tr(cf(ite(Expr,L1,L2),S),cf(C,S)) :- beval(not(Expr),S), at(L2,C).`
`goto L;`	`tr(cf(goto(L),S),cf(C,S)) :- at(L,C).`
`Id = F(ArgList);`	`tr(cf(call(F,ArgList,Id,Ret),S),cf(goto(Ep),S1)) :- prologue(F,ArgList,S,Id,Ret,Ep,S1).`
`return Expr;`	`tr(cf(ret(Expr),S),cf(C,S1)) :- epilogue(Expr,S,S1,Ret), at(Ret,C).`

Safety Verification

Int:

```
unsafe    :- initial(A), reach(A).
reach(A)  :- tr(A,B), reach(B).
reach(A)  :- error(A).

initial((X,Y,N)) :- X=0, Y=0
error((X,Y,N)) :- X>Y.
```

+ clauses for tr (i.e., the interpreter of the language C)
+ clauses for at (i.e., the given C program P)

Theorem: Program P is safe iff the atom unsafe does not belong to the least model $M(\mathsf{Int})$ of the CLP program Int.

Program Specialization

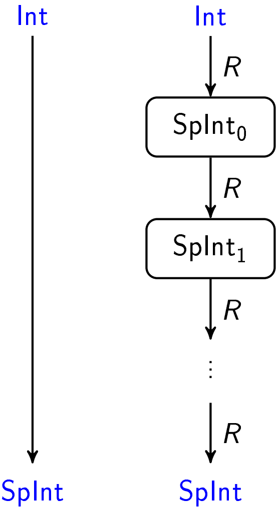
Program specialization is a program manipulation technique whose objective is the adaptation of a program to a context of use.

It is based on transformation rules.

It allows an agile development of verification tools because:

- it is parametric w.r.t. languages and logics
- it allows the composition of various program transformations

Specialize



- transformation rules for specialization:

  $R \in \{$ Definition Introduction,
  Unfolding,
  Folding,
  Clause Removal $\}$

- rules are semantic preserving:

  unsafe $\in M(\text{Int})$ *iff* unsafe $\in M(\text{SpInt})$

- specialization strategy :

  $($ Unfolding ; Clause Remov ; Def Intro ; Folding $)^*$

Rules for Specializing CLP Programs

R1. Definition Introduction : $\mathtt{newp}(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2. Unfolding : $p(X_1, \ldots, X_n) \leftarrow c \wedge \underline{q(X_1, \ldots, X_n)}$

$$\underline{q(X_1, \ldots, X_n)} \leftarrow \underline{d_1 \wedge A_1}, \ \ldots, \underline{q(X_1, \ldots, X_n)} \leftarrow \underline{d_m \wedge A_m}$$

yields

$$p(X_1, \ldots, X_n) \leftarrow c \wedge d_1 \wedge A_1, \ \ldots, p(X_1, \ldots, X_n) \leftarrow c \wedge d_m \wedge A_m$$

R3. Folding : $p(X_1, \ldots, X_n) \leftarrow c \wedge \underline{A}$

$$\underline{q(X_1, \ldots, X_n)} \leftarrow d \wedge \underline{A} \quad \text{and} \quad c \rightarrow d$$

yields

$$p(X_1, \ldots, X_n) \leftarrow c \wedge \underline{q(X_1, \ldots, X_n)}$$

R4. Clause Removal :

if c is unsatisfiable or $\big(p(X_1, \ldots, X_n) \leftarrow d$ and $c \rightarrow d\big)$,

then remove $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$

Specialization strategy

Let `unsafe` be a clause of the form: `unsafe :- Body.`

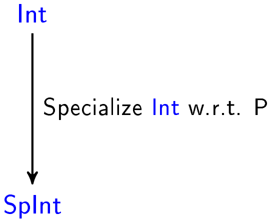
Specialize(Int, `unsafe`) {
 SpInt $= \emptyset$;
 Def $= \{$`unsafe`$\}$;
 while $\exists q \in$ Def **do**
  Unf $=$ Clause Removal(Unfold(q));
  Def $=$ (Def $- \{q\}) \cup$ Define(Unf);
  SpInt $=$ SpInt $\cup$ Fold(Unf, Def);
}

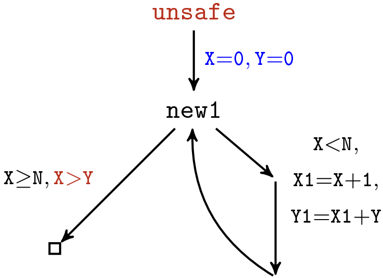
- P is safe *iff* `unsafe` $\notin M$(SpInt),
- Define realizes different specializations (w.r.t. P, φ_{init}, and φ_{error}),
- generalizations in Define ensure termination.

Removal of the Interpreter

Compile away the C interpreter, i.e., remove all references to:

- tr (i.e., the operational semantics of C)
- at (i.e., the encoding of P)





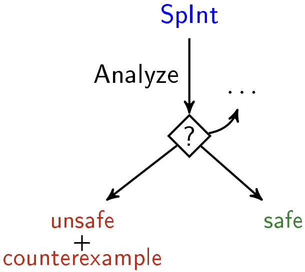
SpInt:

```
unsafe :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

Checking safety of P

Analyze SpInt to check safety of P:

- P is safe iff `unsafe` $\notin M(\text{SpInt})$,
- checking whether or not `unsafe` belongs to $M(\text{SpInt})$ is undecidable,



looking for constrained facts:

- no constrained facts implies $M(\text{SpInt}) = \emptyset$,
- $M(\text{SpInt}) = \emptyset$ implies that P is safe,
- very efficient,
- precision achieved by iterated specialization.

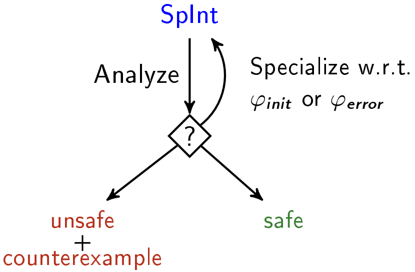
Analyze SpInt

We only look for constrained facts in SpInt:

```
unsafe :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

SpInt has a constrained fact for new1

At this point we cannot show that unsafe does not hold.



We need further specializations.

Specialize SpInt w.r.t. φ_{init}

The output of Specialize, i.e., SpInt

```
unsafe :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

can be viewed as a transition system:

```
initial((new1,X,Y,N)) :- X=0, Y=0.
tr((new1,X,Y,N),(new1,X1,Y1,N)) :- X<N, X1=X+1, Y1=X1+Y.
error((new1,X,Y,N)) :- X≥N, X>Y.
```

By specializing:

```
unsafe :- initial(A), reach(A).          clauses for initial
reach(A) :- tr(A,B), reach(B).     +                 tr
reach(X) :- error(A).                                error
```

w.r.t. unsafe, we propagate the constraint X=0, Y=0 of the initial configuration φ_{init}.

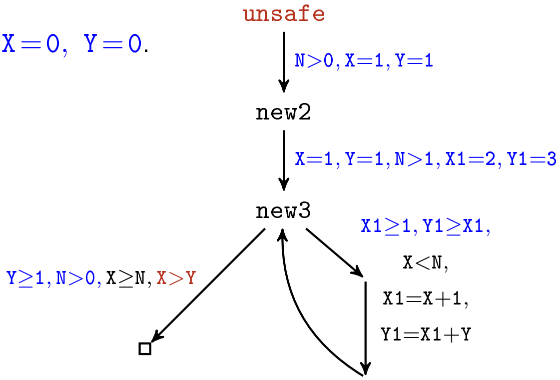
Propagation of the initial configuration

Propagation of the constraint X = 0, Y = 0.

SpInt

↓ Specialize w.r.t. φ_{init}

(new) SpInt




```
unsafe :- N > 0, X1 = 1, Y1 = 1, new2(X1, Y1, N).
new2(X, Y, N) :- X = 1, Y = 1, N > 1, X1 = 2, Y1 = 3, new3(X1, Y1, N).
new3(X, Y, N) :- X1 ≥ 1, Y1 ≥ X1, X < N, X1 = X + 1, Y1 = X1 + Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y ≥ 1, N > 0, X ≥ N, X > Y.
```

Specialize SpInt w.r.t. φ_{error}

The output of Specialize, i.e., SpInt

```
unsafe :- N>0, X1=1, Y1=1, new2(X1,Y1,N).
new2(X,Y,N) :- X=1,Y=1,N>1,X1=2,Y1=3, new3(X1,Y1,N).
new3(X,Y,N) :- X1≥1, Y1≥X1, X<N,X1=X+1, Y1=X1+Y, new3(X1,Y1,N).
new3(X,Y,N) :- Y≥1, N>0, X≥N, X>Y.
```

can be viewed as a transition system:

```
initial((new1,X,Y,N)) :- N>0, X1=1, Y1=1,.
tr((new2,X,Y,N),(new3,X1,Y1,N)) :- X=1,Y=1,N>1,X1=2,Y1=3.
tr((new3,X,Y,N),(new3,X1,Y1,N)) :-
    X1≥1,Y1≥X1,X<N,X1=X+1,Y1=X1+Y.
error((new3,X,Y,N)) :- Y≥1, N>0, X≥N, X>Y.
```

In order to propagate the constraint of the error configuration φ_{error} we reverse the direction of the reachability relation `reach`.

Program Reversal

By specializing

SpInt:

```
unsafe :- initial(A), reach(A).
reach(A) :- tr(A,B), reach(B).
reach(X) :- error(A).
```

w.r.t. `unsafe`, we propagate the constraint of the initial configuration φ_{init}.

By specializing

RevSpInt:

```
unsafe :- error(A), reach(A).
reach(B) :- tr(A,B), reach(A).
reach(X) :- initial(A).
```

w.r.t. `unsafe`, we propagate the constraint of the error configuration φ_{error}.

`unsafe` $\in M(\text{SpInt})$ iff `unsafe` $\in M(\text{RevSpInt})$

Propagation of the error configuration

Propagation of the constraint X > Y.

```
unsafe :- N>0, X1=1, Y1=1, new2(X1,Y1,N).
new2(X,Y,N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1,Y1,N).
new3(X,Y,N) :- X1≥1, Y1≥X1, X<N,X1=X+1, Y1=X1+Y, new3(X1,Y1,N).
new3(X,Y,N) :- Y≥1, N>0, X≥N, X>Y.
```

SpInt

↓ Reverse

RevSpInt

↓ Specialize w.r.t. φ_{error}

(new) SpInt

unsafe

↓ Y ≥ 1, N > 0, X ≥ N, X > Y

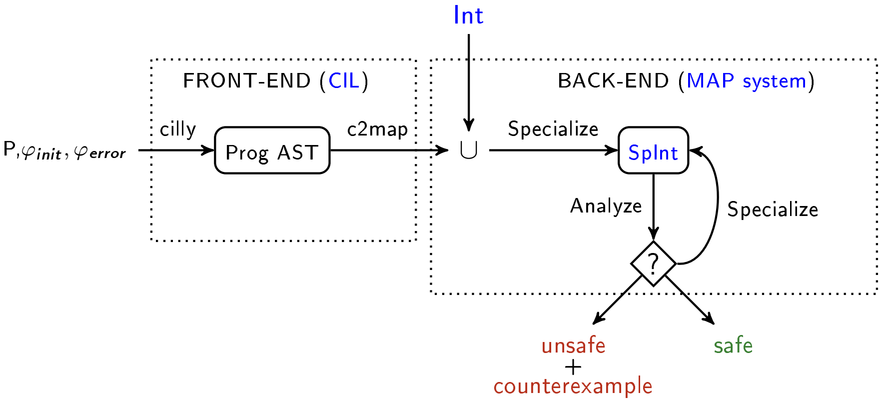
new4

```
unsafe :- Y≥1, N>0, X≥N, X>Y, new4(X,Y,N).
```

Software Model Checker Architecture

Fully automatic Software Model Checker for proving safety of C programs.

* CIL (C Intermediate Language)
  http://http://kerneis.github.com/cil/
* MAP Transformation System
  http://map.uniroma2.it/mapweb



Experimental result

Program	MAP(a)		MAP(b)		ARMC	HSF(C)	TRACER	
	n		*n*				*SPost*	*WPre*
barber1	1	13.71	2	26.43	414.01	0.59	7.00	5.17
berkeley	1	1.57	2	1.53	11.28	0.26	-	1.00
efm	3	6.48	2	4.04	31.17	0.51	2.43	2.68
ex1	1	0.03	2	0.40	1.69	0.22	-	1.39
f1a	2	0.17	1	0.07	-	0.21	-	1.97
heapSort	1	8.16	2	13.51	39.66	0.35	-	-
heapSort1	1	3.01	2	9.58	20.55	0.26	-	-
interp	1	0.12	2	0.28	11.41	0.19	-	2.92
lifo	1	20.56	2	15.59	126.54	0.54	-	7.45
p2	1	14.75	1	-	-	0.77	-	-
re1	1	0.23	1	0.08	-	0.19	-	-
selectSort	3	1.96	6	3.26	24.97	0.25	-	-
singleLoop	3	0.35	2	0.28	-	-	-	56.57
substring	2	0.16	1	0.20	472.32	40.51	-	-
tracerP	1	0.01	1	0.07	-	-	1.04	1.03
...	...	...	...	...	...	...	...	...
#verified programs		20 (9)		19 (15)	13	18	4	14
total time		353.29		209.94	1971.10	69.42	23.33	206.78

Time (in seconds). '-' means 'unable to verify within 10 min'. *n* is the number of specializations performed by the MAP system (after removal of the interpreter).

Conclusions and Future work

Software Model Checking framework, which is parametric w.r.t.:

- the language of the programs to be verified,
- the logic of the property to be checked.

Future Work:

- more features of the C language (arrays, pointers, etc.),
- more complex properties (e.g., liveness properties)
- different languages (e.g., Java and C#)
  to deal with object-oriented features and concurrency.