Verification of Programs by Combining Iterated Specialization with Interpolation

Emanuele De Angelis[1,3], Fabio Fioravanti[1], Jorge A. Navas[2], and Maurizio Proietti[3]

[1]University of Chieti-Pescara, Italy
[2]NASA Ames Research Center, USA
[3]CNR - Istituto di Analisi dei Sistemi ed Informatica, Italy

HCVS 2014
Vienna, July 17, 2014

Given the program *increment* and the specification φ

```
while(*) {
  x=x+y;
  y=y+1;
}
```

$\{x=1 \wedge y=0\}$ *increment* $\{x \geq y\}$

(A) generate the verification conditions (VCs)

1.	$x=1 \wedge y=0 \rightarrow P(x,y)$	Initialization
2.	$P(x,y) \rightarrow P(x+y, y+1)$	Loop invariant
3.	$P(x,y) \rightarrow x \geq y$	Exit

(B) prove they are satisfiable

If satisfiable then the Hoare triple holds.

Motivations (2)

- Constraint Logic Programming (CLP) is a metalanguage for representing
  programs and their semantics,
  properties and their proof rules
  i.e., for representing the Verification Conditions (VCs)

1.	$x=1 \land y=0 \rightarrow P(x,y)$	Initialization
2.	$P(x,y) \rightarrow P(x+y, y+1)$	Loop invariant
3.	$P(x,y) \rightarrow x \geq y$	Exit

The VCs are encoded as a constraint logic program V:

```
1. p(X,Y) :- X=1, Y=0.                     Constrained fact
2. p(X1,Y1) :- X1=X+Y, Y1=Y+1, p(X,Y).     Rule
4. unsafe :- Y>X, p(X,Y).                  Query
```

The VCs are satisfiable iff `unsafe` not in the least model M of V.

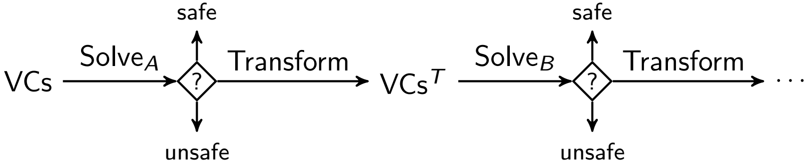
- Constraint Logic Programming (CLP) is a metalanguage for representing
  programs and their semantics,
  properties and their proof rules
  i.e., for representing the Verification Conditions (VCs)

  Methods for proving the satisfiability of CLP/CHC VCs:
  - CounterExample Guided Abstraction Refinement (CEGAR), Interpolation, Satisfiability Modulo Theories
    [Bjørner et al., Duck et al., Rybalchenko et al., Rümmer et al.]
  - Symbolic execution of CLP
    [Jaffar et at.]
  - Static Analysis and Transformation of CLP
    [Gallagher et al., Albert et al., Fioravanti et al.]

Motivations (2)

- Constraint Logic Programming (CLP) is a metalanguage for representing
  programs and their semantics,
  properties and their proof rules
  i.e., for representing the Verification Conditions (VCs)

- Program Transformation is a technique that
  changes the syntax of a program,
  preserves its semantics
  i.e., for *passing* information between Solvers



$$prop \in M(\text{VCs}) \text{ iff } prop \in M(\text{VCs}^T)$$

In this work ... (Outline of the Talk)

Verification method for program safety that combines

- Program Specialization

  unfold/fold transformations + widening

- Interpolating Horn Clause (IHC) solving

  top-down evaluation + interpolation

by exploiting the common Horn Clause representation of the problem.

Hence, *combining* the effect of interpolation to the effect of widening.

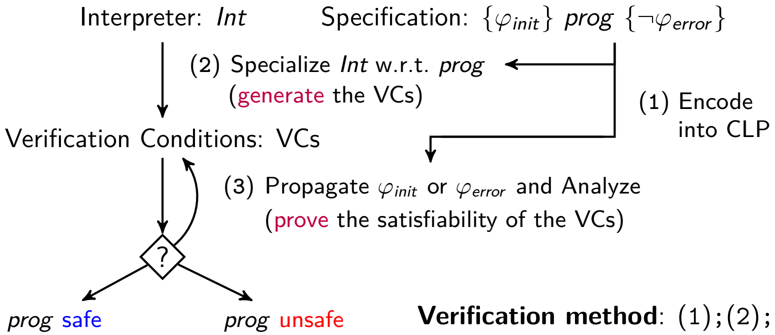
Specialization and Interpolation phases can be:

- *iterated*, and also
- *combined* with other transformations that change the direction of propagation of the constraints:
  - forward from the program preconditions, or
  - backward from the error conditions.

The Transformation-based Verification Method

Program transformation of Constraint Logic Programs (CLP) to:

- generate the Verification Conditions (VCs)
- prove the satisfiability of the VCs



Verification method: $(1);(2);(3)^+$

Encoding Partial Correctness into CLP

Given the specification $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$

Definition (The interpreter *Int*)

```
unsafe :- initConf(X), reach(X).           | X satisfies φinit
reach(X) :- tr(X,Y), reach(Y).
reach(X) :- errorConf(X).                  | X satisfies φerror
```

+ clauses for `tr` (the semantics of the programming language)

A program *prog* is unsafe w.r.t. φ_{init} and φ_{error}
if from an initial configuration satisfying φ_{init}
it is possible to reach a final configuration satisfying φ_{error}.
Otherwise, program *prog* is safe.

Theorem

prog is safe iff `unsafe` $\notin M(Int)$ *(the least model of Int)*

Encoding the Verification Problem into CLP

Given the program *increment* and the specification φ

```
while(*) {
  x=x+y;
  y=y+1;
}
```

$\{x=1 \wedge y=0\}$ *increment* $\{x \geq y\}$

CLP encoding of program *increment*

A set of at(label, command) facts.
while commands are replaced by ite and goto.

$\mathtt{at}(\ell_0, \mathtt{ite}(\mathtt{nondet}, \ell_1, \ell_h)).$
$\mathtt{at}(\ell_1, \mathtt{asgn}(\mathtt{x}, \mathtt{plus}(\mathtt{x}, \mathtt{y}))).$
$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{y}, \mathtt{plus}(\mathtt{y}, 1))).$
$\mathtt{at}(\ell_3, \mathtt{goto}(\ell_0)).$
$\mathtt{at}(\ell_h, \mathtt{halt}).$

CLP encoding of φ_{init} and φ_{error}

$\mathtt{initConf}(\ell_0, \mathtt{X}, \mathtt{Y})$:- $\mathtt{X}=1, \mathtt{Y}=0.$
$\mathtt{errorConf}(\ell_h, \mathtt{X}, \mathtt{Y})$:- $\mathtt{X}<\mathtt{Y}.$

Transforming CLP Programs

$$\boxed{P} \xrightarrow{R} P_1 \xrightarrow{R} P_2 \xrightarrow{R} \cdots \xrightarrow{R} \boxed{TransfP}$$

Rule-based program transformation

- transformation rules:

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

- the transformation rules preserve the least model:

Theorem

$\texttt{unsafe} \in M(P)$ *iff* $\texttt{unsafe} \in M(TransfP)$

- the rules must be guided by a strategy.

The Unfold/Fold Transformation Strategy

Transform(P)

$TransfP = \emptyset$;

Defs $= \{$`unsafe :- initConf(X), reach(X)`$\}$;

while $\exists q \in$ Defs **do**

`% execute a symbolic evaluation step (resolution)`

Cls $=$ Unfold(q);

`% remove unsatisfiable and subsumed clauses`

Cls $=$ ClauseRemoval(Cls);

`% introduce new predicates (e.g., a loop invariant)`

Defs $=$ (Defs $- \{q\}$) $\cup$ Define(Cls);

`% match a predicate definition`

$TransfP = TransfP \cup$ Fold(Cls, Defs);

od

Generating Verification Conditions

The specialization of *Int* w.r.t. *prog* removes all references to:

- `tr` (i.e., the operational semantics of the imperative language)

```
L: goto L1;
tr(cf(cmd(L,goto(L1)),S), cf(C,S)) :- at(L1,C).
```

- `at` (i.e., the encoding of *prog*)

The Specialized Interpreter for *increment* (Verification Conditions)

```
unsafe :- X=1, Y=0, new1(X,Y).
new1(X,Y) :- X=X+Y, Y=Y+1, new1(X,Y).
new1(X,Y) :- X<Y.
```

New predicates correspond to a subset of the *program points*:

```
new1(X,Y) :- reach(cf(cmd(0,ite(...)),
              [[int(x),X],[int(y),Y]])).
```

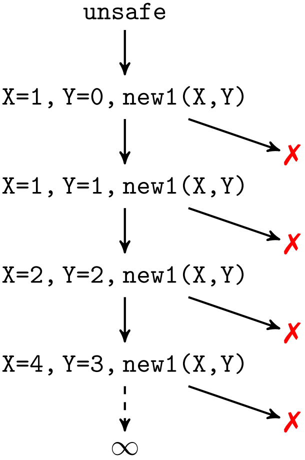
Proving Satisfiability of VCs (1)

Satisfiability of a set of clauses can be reduced to the standard top-down query evaluation.

```
unsafe :- X=1, Y=0, new1(X,Y).
new1(X,Y) :- X=X+Y, Y=Y+1, new1(X,Y).
new1(X,Y) :- X<Y.
```

the recursive predicate `new1`, generates an infinite derivation for `unsafe`.

top-down evaluation with *tabling* (i.e. memoing of partial answers) does not terminate.



Failure Tabled CLP (FTCLP) (2)

Interpolating Horn Clause (IHC) Solver:

- interpolation provides learned facts from failure that can be used for pruning search.

Failure Tabled CLP (FTCLP): [Navas et al.]

- an interpolating Horn Clause (IHC) Solver
- execute a set of CLP clauses top-down while labelling nodes in the derivation tree with interpolants:
  1. Whenever a loop is detected its execution stops, and it backtracks to an ancestor choice point,
  2. After completion of a subtree, the tabling mechanism will attempt at proving that the predicate where the execution was frozen can be subsumed by any of its ancestors using an interpolant as the subsumption condition
  3. If it fails then its execution is re-activated and the process continues.

```
1. unsafe :- X=1, Y=0, new1(X,Y).
2. new1(X,Y) :- X=X+Y, Y=Y+1, new1(X,Y).
3. new1(X,Y) :- X<Y.
```

(a) freeze the execution of the recursive clause 2

```
unsafe
  ↓
X=1, Y=0, new1(X,Y)
  ↓                  ↘ ✗
X1=Y+X, Y1=Y+1, new1(X1,Y1)
```

(b) learn X≥Y from the failed derivation: X=1, Y=0, X<Y (compute an interpolant between X=1, Y=0 and X<Y)

(c) check if X≥Y is an inductive invariant

Unfortunately, X≥Y, X1=X+Y, Y1=Y+1 $\not\models$ X1≥Y1

X≥Y is not an inductive invariant

Transforming Verification Conditions

Program transformation:

- propagates constraints,
- introduces predicate definitions (i.e., program invariants)

Use of generalization operators:

- to ensure the termination of the transformation,
- to generate program invariants,

... two somewhat conflicting requirements:

- efficiency, to introduce as few definitions as possible,
- precision, to prove as many properties as possible.

Generalization operators add new constraints to predicate definitions that might make the top-down (or bottom-up) evaluation terminating.

Constraint Generalizations

Definitions are arranged as a tree:

```
    unsafe :- i, A
      /        \
   ...        newp :- c, B      ancestor definition
                |
              newq :- d, B      candidate definition

              newr :- g, B      generalized definition
```

d → g

Generalization operators based on widening and convex-hull.

Propagating the initial configuration φ_{init}

The verification conditions are specialized w.r.t. φ_{init}.

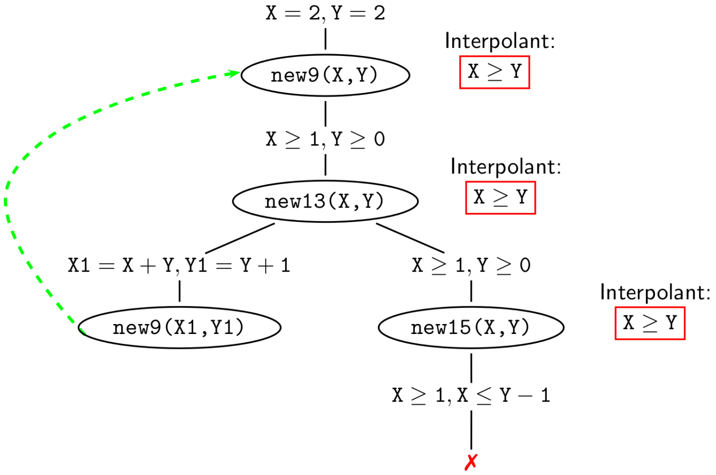
Specialized Verification Conditions for *increment*

... *propagating* the constraint $X=1,\ Y=0$.

```
unsafe :- X=1, Y=0, new4(X,Y).
new4(X,Y) : X=1, Y=0, X1=1, Y1=1, new5(X1,Y1).
new5(X,Y) : X=1, Y≥0, new8(X,Y).
new8(X,Y) :- X=1, X1=Y+1, X1≥1, Y1=X1, new9(X1,Y1).
new8(X,Y) :- X=1, Y≥0, new10(X,Y).
new10(X,Y) :- X=1, Y≥2.
new9(X,Y) :- X≥1, Y≥0, new13(X,Y).
new13(X,Y) :- X1=X+Y, Y1=Y+1, new9(X1,Y1).
new13(X,Y) :- X≥1, Y≥0, new15(X,Y).
new15(X,Y) :- X≥1, X≤Y-1.
```

The transformation adds new constraint $X\geq 1, Y\geq 0$,
so that FTCLP solver terminates.

Execution of Recursive CHCs



$\mathtt{X} = 2, \mathtt{Y} = 2 \models \mathtt{X} \geq \mathtt{Y}$ (by definition of interpolation)

$\mathtt{X} \geq \mathtt{Y}, \mathtt{X} \geq 1, \mathtt{Y} \geq 0, \mathtt{X1} = \mathtt{X} + \mathtt{Y}, \mathtt{Y1} = \mathtt{Y} + 1 \models \mathtt{X1} \geq \mathtt{Y1}$

Discovering program invariants

Program transformation and FTCLP improve on infinite failure.

- generalization operators may discover invariants by looking at the history of the computation
  e.g., from `X=1, Y=0` and `X=2, Y=1` (one loop execution)
  by generalization we derive `X>=1, Y>=0`
- interpolation discovers invariants by looking at failed executions
  e.g., from `X=1, Y=0` and `X<Y` we derive `X>=Y`.

If the invariants are not strong enough to prove the correctness of the program, we iterate the transformation process.

Program Reversal

By specializing

Int:

```
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

RevInt:

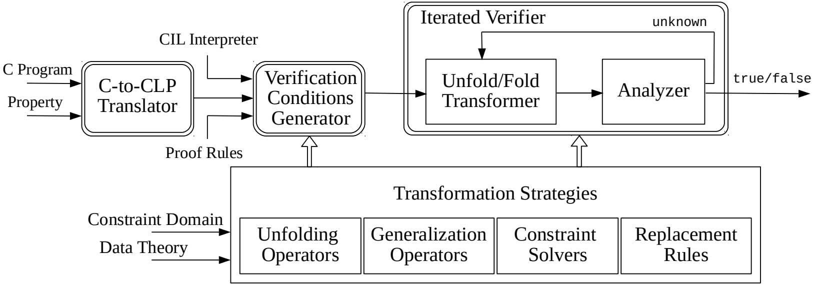
```
unsafe :- error(A), reach(A).
reach(B) :- tr(A,B), reach(A).
reach(X) :- initial(A).
```

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

$$\texttt{unsafe} \in M(\mathsf{Int}) \text{ iff } \texttt{unsafe} \in M(\mathsf{RevInt})$$

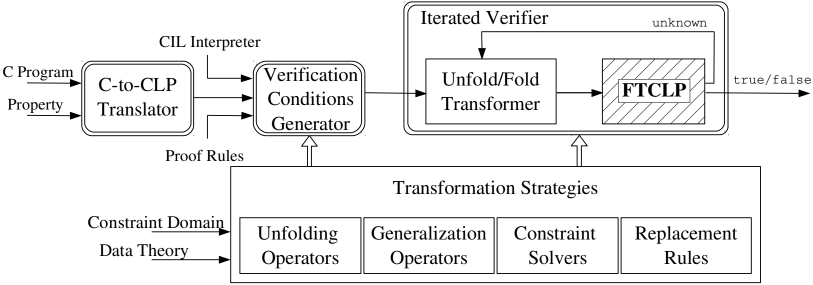
A Tool for Verifying Programs through Transformations

- CIL (C Intermediate Language) by Necula et al.
- MAP Transformation System by the MAP group



Available at: `http://map.uniroma2.it/VeriMAP`

The architecture of the **VeriMAP** tool with **FTCLP**.



The FTCLP solver is implemented using :

- Ciao prolog system
- MathSAT (for the interpolants generation)

Available at: `http://code.google.com/p/ftclp`

Experimental evaluation

	FTCLP	VeriMAP$_M$	VeriMAP$_M$ + FTCLP	VeriMAP$_{PH}$	VeriMAP$_{PH}$ + FTCLP
answers	116	128	160	178	182
crashes	5	0	2	0	0
timeouts	95	88	54	38	34
total time	12470.26	11285.77	9714.41	5678.09	6537.17
average time	107.50	88.17	60.72	31.90	35.92

Table : Verification results using VeriMAP, FTCLP, and the combination of VeriMAP and FTCLP. The timeout limit is two minutes. Times are in seconds.

Iteration	VeriMAP$_M$	VeriMAP$_M$ + FTCLP	VeriMAP$_{PH}$	VeriMAP$_{PH}$ + FTCLP
1	74	119	104	136
2	45	38	54	34
3	7	2	10	5
4	2	1	8	3
5	0	0	2	4

Table : Number of definite answers computed by VeriMAP and by the combination of VeriMAP and FTCLP within the first five iterations.

Conclusions and Future Work

- Parametric verification framework (semantics and logic)
  - CLP as a metalanguage
  - Semantics preserving transformations to:
    - iterate specialization and analysis, and
    - pass information between verifiers

  thereby resulting in an incremental verification process.
- We instantiated the verification framework by integrating:
  - an Iterated Specialization tool (VeriMAP), and
  - an Interpolating Horn Clauses Solver (FTCLP).
- Future work: combine these tools in a more synergistic way
  - leverage the partial information FTCLP discovers and integrate it into the Specialized program,
  - refining the generalization step by using the interpolants computed by FTCLP, and
  - use interpolation during the transformation process.