VeriMAP
A Tool for Verifying Programs through Transformations

Emanuele De Angelis, Fabio Fioravanti,
Alberto Pettorossi, and Maurizio Proietti

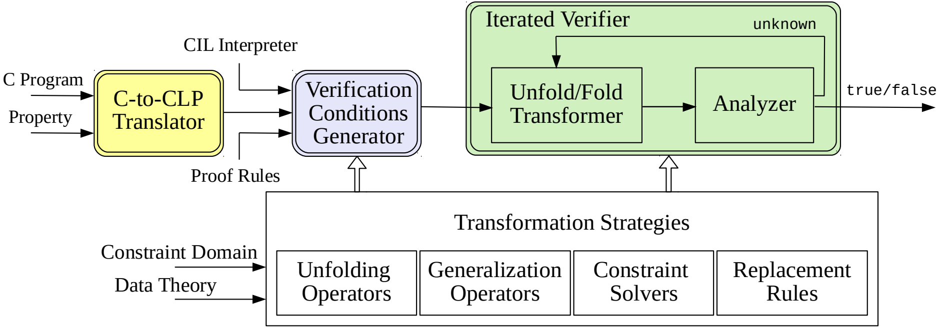
University of Chieti – Pescara ‘G. d’Annunzio’,
University of Rome ‘Tor Vergata’, and
IASI – CNR of Rome

Milano, 26 September 2014

What is VeriMAP?

- a tool for the verification of **safety properties** of C programs manipulating integers and integer arrays
- based on **Constraint Logic Programs** (CLP) as a **metalanguage** for representing:
  - the operational semantics of the C language
  - the proof rules for safety
  - the C program to be verified
  - the safety property to be checked
- *satisfiability preserving* **transformations of CLP programs** for:
  - generating Verification Conditions
  - checking their satisfiability

Tool Architecture



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

Verification of Safety Properties

Given the specification $\{\varphi_{init}\}$ *CProg* $\{\psi\}$, define $\varphi_{error} \equiv \neg\psi$

```
int x, y, n;
while(x<n) {
    x=x+1;
    y=y+2;
}
```

Initial and error properties

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

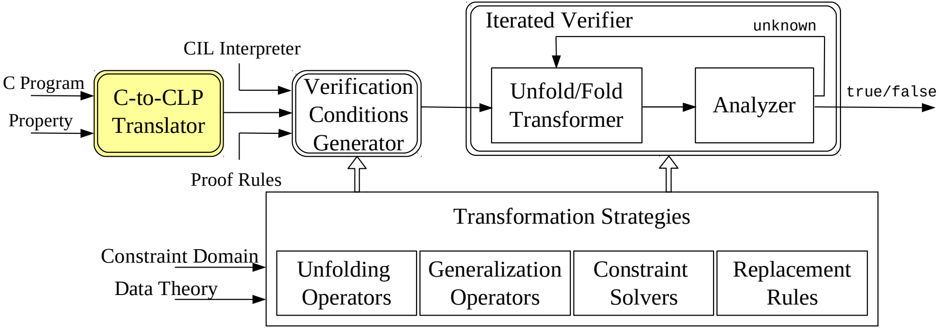
$\varphi_{error}(\mathtt{x},\mathtt{y},\mathtt{n}) \equiv \mathtt{y} > 2\mathtt{x}$

A program is incorrect w.r.t. φ_{init} and φ_{error} iff from an initial configuration satisfying φ_{init} it is possible to reach a final configuration satisfying φ_{error}.

Step 1: C-to-CLP - Translating C programs into CLP

Construct the CLP encoding of

- the C Program *CProg* as a set of facts `at(Label,Command)`
- the Property $\langle \varphi_{init}, \varphi_{error} \rangle$ as constrained facts



C-to-CLP translator

- First the C program is preprocessed using CIL.
  - `while`'s and `for`'s are translated into equivalent commands that use `if-else`'s and `goto`'s.
- Then, for each program command, C-to-CLP generates a CLP fact of the form `at(L, C)`, where `C` and `L` represent the command and its label.

1. ℓ_0 : `if (x<n) goto` ℓ_1; `else goto` ℓ_h;
2. ℓ_1 : `x=x+1;`
3. ℓ_2 : `y=y+2;`
4. ℓ_3 : `goto` ℓ_0;
5. ℓ_h : `halt;`

```
1. at(l0,ite(less(x,n),l1,lh)).
2. at(l1,asgn(x,expr(plus(x,1)),l2)).
3. at(l2,asgn(y,expr(plus(y,2)),l3)).
4. at(l3,goto(l0)).
5. at(lh,halt).
```

- Also facts for the initial and error properties are generated:

```
phiInit(cf(...,[(x,X),(y,Y),(n,N)])) :- X=0, Y=0, N>=0.
phiError(cf(...,[(x,X),(y,Y),(n,N)])) :- Y>2*X.
```

The CLP interpreter *Int*

Proof rules for safety

```
incorrect :- initial(X), phiInit(X), reach(X).
reach(X) :- tr(X,Y), reach(Y).
reach(X) :- final(X), phiError(X).
```

Operational semantics of the programming language

```
tr(cf(Lab1,Cmd1),cf(Lab2,Cmd2)) :- ···
```

e.g., operational semantics of the conditional command

`L: if(Expr) {`	`tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-`
`  L1: ...`	`beval(Expr,S),` *expression is true*
`}`	`at(L1,C).` *next command*
`else`	`tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-`
`  L2: ...`	`beval(not(Expr),S),` *expression is false*
`}`	`at(L2,C).` *next command*

Correctness of Encoding:

CProg is correct iff `incorrect` $\notin M(Int)$ (the least model of *Int*)

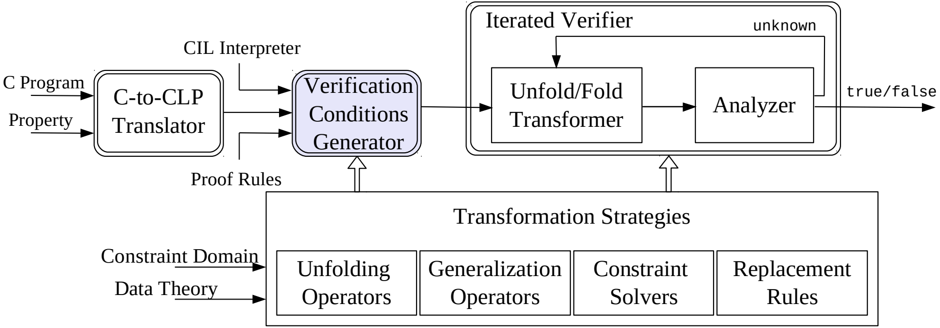
Step 2: Generating Verification Conditions

Generate the Verification Conditions (VCs) by **specializing** the CLP interpreter *Int* (CIL Interpreter + Proof Rules) w.r.t. the CLP encoding of the C program *CProg* .

All references to

- tr (operational semantics of the C language)
- at (encoding of the C program *CProg*)

are removed.



Rule-based Program Transformation

$$
\begin{array}{c}
\boxed{P} \\
\downarrow R \\
P_1 \\
\downarrow R \\
P_2 \\
\downarrow R \\
\vdots \\
\downarrow R \\
\boxed{TransfP}
\end{array}
$$

- transformation **rules**:

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

- the transformation rules

  change the **syntax** of a program
  preserve its **least model semantics**.

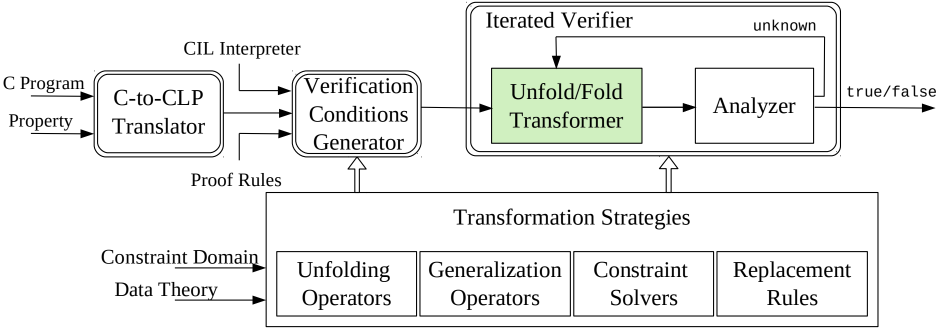
  $\texttt{incorrect} \in M(P)$ *iff* $\texttt{incorrect} \in M(TransfP)$

- the rules are guided by a **strategy**.

Step 3: Transforming the VCs

Transform the VCs by propagating either

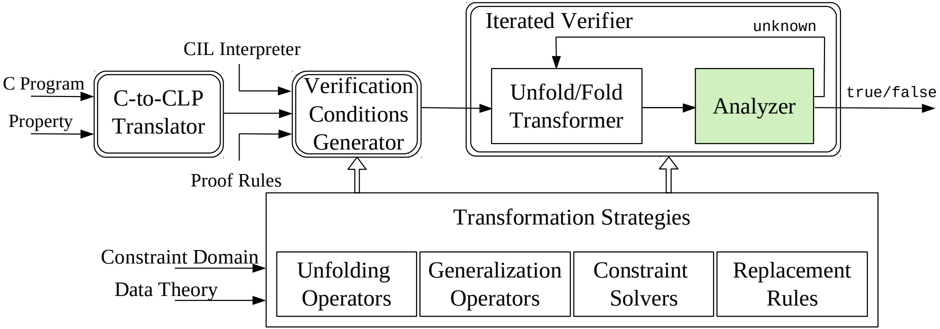
- the constraint encoded by `phiInit` (φ_{init}) or
- the constraint encoded by `phiError` (φ_{error})



Step 4: Checking satisfiability of the VCs

Analyze the CLP program representing the transformed VCs

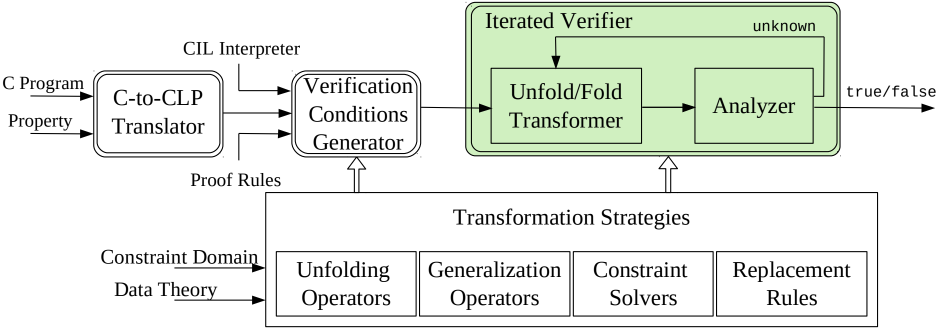
- *CProg* correct if no constrained facts appear in the VCs.
- *CProg* incorrect if the fact `incorrect.` appears in the VCs.



Precision achieved by iteration:

- reverse the direction of the state-space exploration
- transform and analyze

(i.e., alternate the propagation of φ_{init} and φ_{error})



Experimental Evaluation - Integer Programs

216 examples taken from: DAGGER, TRACER, InvGen, and TACAS 2013 Software Verification Competition.

		VeriMAP	ARMC	HSF(C)	TRACER
1	*correct answers*	185	138	160	103
2	safe problems	154	112	138	85
3	unsafe problems	31	26	22	18
4	*incorrect answers*	0	9	4	14
5	false alarms	0	8	3	14
6	missed bugs	0	1	1	0
7	*errors*	0	18	0	22
8	*timed-out problems*	31	51	52	77
9	*total time*	10717.34	15788.21	15770.33	23259.19
10	*average time*	57.93	114.41	98.56	225.82

- ARMC [Podelski, Rybalchenko PADL 2007]
- HSF(C) [Grebenshchikov et al. TACAS 2012]
- TRACER [Jaffar, Murali, Navas, Santosa CAV 2012]

Experimental evaluation - Array Programs

Program	$Gen_{W,\mathcal{I},\Cap}$	$Gen_{H,\mathcal{V},\subseteq}$	$Gen_{H,\mathcal{V},\Cap}$	$Gen_{H,\mathcal{I},\subseteq}$	$Gen_{H,\mathcal{I},\Cap}$
bubblesort-inner	0.9	*unknown*	*unknown*	*unknown*	1.52
copy-partial	*unknown*	*unknown*	3.52	3.51	3.54
copy-reverse	*unknown*	*unknown*	5.25	*unknown*	5.23
copy	*unknown*	*unknown*	5.00	4.88	4.90
find-first-non-null	0.14	0.66	0.64	0.28	0.27
find	1.04	6.53	2.35	2.33	2.29
first-not-null	0.11	0.22	0.22	0.22	0.22
init-backward	*unknown*	1.04	1.04	1.03	1.04
init-non-constant	*unknown*	2.51	2.51	2.47	2.47
init-partial	*unknown*	0.9	0.89	0.9	0.89
init-sequence	*unknown*	4.38	4.33	4.41	4.29
init	*unknown*	1.00	0.97	0.98	0.98
insertionsort-inner	0.58	2.41	2.4	2.38	2.37
max	*unknown*	*unknown*	0.8	0.81	0.82
partition	0.84	1.77	1.78	1.76	1.76
rearrange-in-situ	*unknown*	*unknown*	3.06	3.01	3.03
selectionsort-inner	*unknown*	*time-out*	*unknown*	2.84	2.83
verified	6	10	15	15	17
total time	3.61	21.42	34.76	31.81	38.45
average time	0.60	2.14	2.31	2.12	2.26

Ongoing and Future Work

VeriMAP is an instance of a general transformation-based Verification Framework, which is parametric w.r.t.

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

Experimenting with:

- other properties (e.g., CTL)
- integration with other tools and techniques (e.g., CEGAR)

Extending the interpreter to deal with:

- dynamic data structures (e.g., heaps)
- recursive functions (e.g., big step semantics)
- other programming language features (e.g., concurrency)
- an assertion specification language

Thank you!

http://map.uniroma2.it/VeriMAP/

CLP with array constraints

Array constraints

- $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$
  the `i`-th element of array `a` is `v`
- $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$
  array `b` is equal to array `a` except that its `i`-th element is `v`
- $\mathtt{dim}(\mathtt{a}, \mathtt{n})$
  the dimension of `a` is `n`

Theory of Arrays

Array congruence

$$\text{(AC)}\quad \mathtt{I}=\mathtt{J},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})\ \rightarrow\ \mathtt{U}=\mathtt{V}$$

Read-over-Write

$$\text{(RoW1)}\quad \mathtt{I}=\mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})\ \rightarrow\ \mathtt{U}=\mathtt{V}$$

$$\text{(RoW2)}\quad \mathtt{I}\neq\mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})\ \rightarrow\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$$

Verification Framework