Software Verification and Synthesis using Constraints and Program Transformation

Emanuele De Angelis

University ‘G. d’Annunzio’ of Chieti–Pescara

Convegno Italiano di Logica Computazionale 2015
Genova, 1 July, 2015

Summary

Verification framework

- Sequential Programs (e.g., C programs)
- Formal language: Constraint Logic Programming
- Proof technique: Program Transformation

Implementation and Experimental Results

- the VeriMAP tool

Synthesis framework

- Concurrent Programs (e.g., Peterson algorithm)
- Formal language: Answer Set Programming
- Synthesis technique: Answer Set Solvers

Verification Conditions as CLP Programs

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

$\{\ x=0 \wedge y=0 \wedge n \geq 1\ \}$

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

$\{\ y > x\ \}$

Verification Conditions (VCs) can be encoded as a set of clauses *P*

incorrect :- X=0, Y=0, N ≥ 1, while(X, Y, N).	Initialization
while(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, while(X1,Y1,N).	Loop body
while(X,Y,N) :- X≥N, Y≤X.	Exit

VCs are satisfiable iff `incorrect` *not* in the least model $M(P)$ of P

How to (automatically)

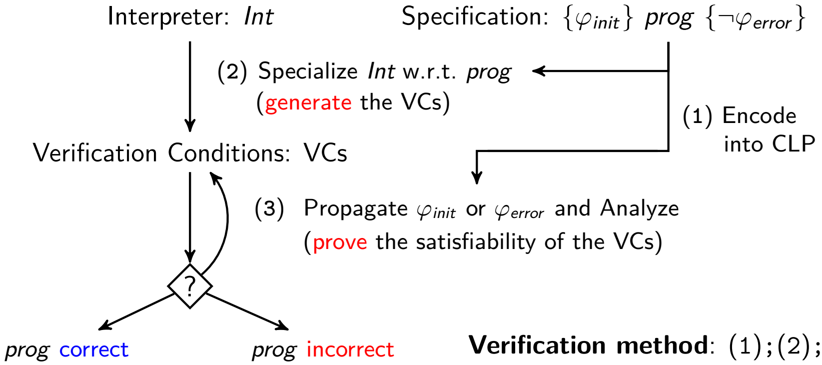
(A) generate the VCs for *prog* ?

(B) prove the satisfiability of the VCs ?

The Transformation-based Verification Method

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: the interpreter *Int*

Proof rules for safety (reachability of error configurations)

```
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 conditionals

```
L: if(Expr){
    L1: ...
  }
  else
    L2: ...
  }
```

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

Theorem (Correctness of Encoding)

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

Encoding program and specification into CLP

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

$\{\ x=0 \wedge y=0 \wedge n \geq 1\ \}$

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

$\{\ y > x\ \}$

CLP encoding of program *prog*

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

```
at(0, ite(less(x,n), 1, h)).
at(1, asgn(x, plus(x, 1))).
at(2, asgn(y, plus(y, 2))).
at(3, goto(0)).
at(h, halt).
```

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

```
phiInit(X,Y,N) :- X=0, Y=0, N≥1.
phiError(X,Y) :- Y≤X.
```

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

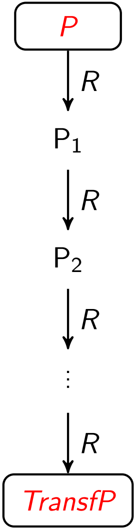
- tr (i.e., the operational semantics of the imperative language)
- at (i.e., the encoding of *prog*)

The Specialized Interpreter for *prog* (Verification Conditions)

```
incorrect :- X=0, Y=0, N≥1, while(X,Y,N).
while(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, while(X1,Y1,N).
while(X,Y,N) :- X≥N, Y≤X.
```

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

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



Rule-based program transformation

- transformation rules:

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

- the transformation rules preserve the least model:

Theorem (Rules are semantics preserving)

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

- the rules must be guided by a strategy.

The Unfold/Fold Transformation Strategy

Transform(P)

$\mathit{TransfP} = \emptyset$;

Defs $= \{$`incorrect :- initial(X), phiInit(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`

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

od

Propagation of φ_{init}

The transformation of the VCs *P*

VCs for *prog* (Specialized interpreter *Int*)

```
incorrect :- X=0, Y=0, N ≥ 1, while(X,Y,N).
while(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, while(X1,Y1,N).
while(X,Y,N) :- X≥N, Y≤X.
```

by propagating the constraint $X=0, Y=0, N \geq 1$,
modifies the structure of *P* and derives the new VCs *TransfP*

Transformed VCs for *prog*

```
incorrect :- X=0, Y=0, N≥1 new1(X,Y,N).
new1(X,Y,N) :- X=0, Y=0, N≥1, X1=1, Y1=2, new2(X1,Y1,N).
new2(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, X1≥1, Y1≥2, new2(X1,Y1,N).
new2(X,Y,N) :- X≥N, Y≤X, Y≥0, N≥1.
```

The fact `incorrect` is not in *TransfP*, we cannot infer that *prog* is incorrect.
A constrained fact is in *TransfP*, we cannot infer that *prog* is correct.

Propagation of φ_{error}

Transformed VCs for *prog* (after the propagation of φ_{init})

```
incorrect :- X=0, Y=0, N≥1, new1(X,Y,N).
new1(X,Y,N) :- X=0, Y=0, N>1, X1=1, Y1=2, new2(X1,Y1,N).
new2(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, X1≥1, Y1≥2, new2(X1,Y1,N).
new2(X,Y,N) :- X≥N, Y≤X, Y≥0, N≥1.
```

Reversed VCs

```
incorrect :- X≥N, Y≤X, Y≥0, N≥1, new2(X,Y,N).
new2(X1,Y1,N) :- X=0, Y=0, N>1, X1=1, Y1=2, new1(X,Y,N).
new2(X1,Y1,N) :- X<N, X1=X+1, Y1=Y+2, X1≥1, Y1≥2, new2(X,Y,N).
new1(X,Y,N) :- X=0, Y=0, N≥1.
```

by propagating φ_{error}, that is, the constraint X ≥ N, Y ≤ X, Y ≥ 0, N ≥ 1.

Transformed VCs for *prog* (after the propagation of φ_{error})

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

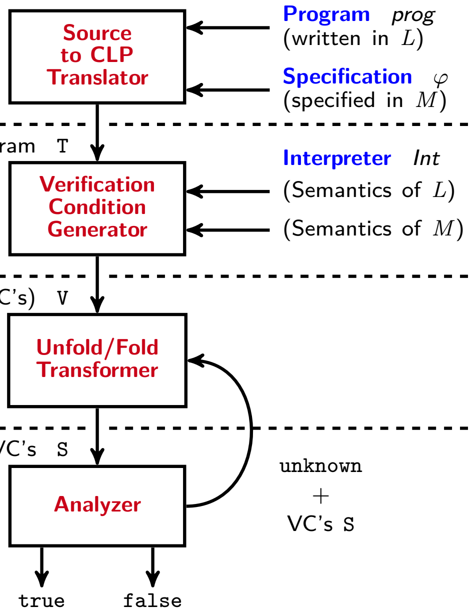
No facts: *prog* is correct.

Step (0) Translate `Prog` and φ into CLP +

Step (1) Specialize *Int* w.r.t. `T` (Removal of the Interpreter)

Step (2) Transform verification conditions w.r.t. φ

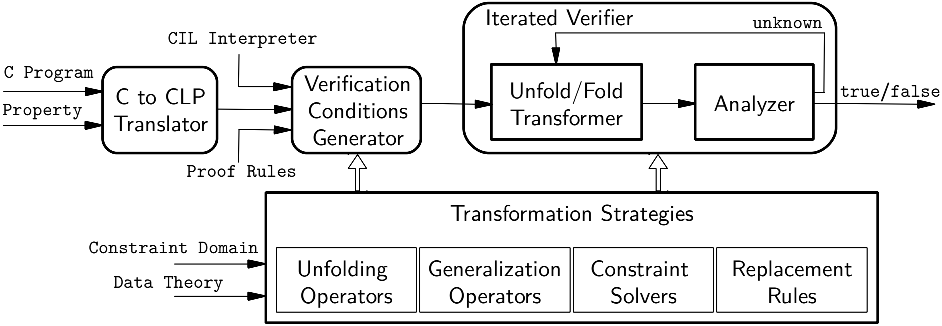
Step (3) Check whether or not φ holds in `Q`



VeriMAP: A Tool for Verifying Programs through Transformations
http://map.uniroma2.it/VeriMAP/

Fully automatic software model checker for C programs.

- **CIL** (C Intermediate Language) by Necula et al.
- **MAP Transformation System** by the MAP group (IASI-CNR, ‘G. d’Annunzio’ and ‘Tor Vergata’ Universities)



Experimental Evaluation - Integer Programs

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

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

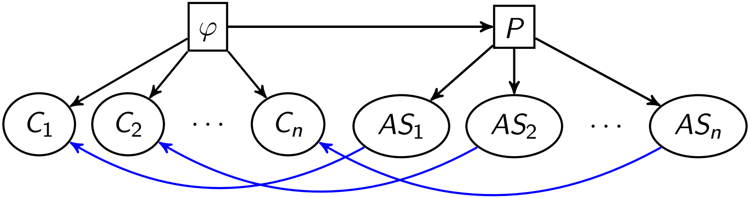
		VeriMAP	ARMC	HSF(C)	TRACER
1	*correct answers*	185	138	160	103
2	safe problems	154	112	138	85
3	unsafe problems	31	26	22	18
4	*incorrect answers*	0	9	4	14
5	false alarms	0	8	3	14
6	missed bugs	0	1	1	0
7	*errors*	0	18	0	22
8	*timed-out problems*	31	51	52	77
9	*total time*	10717.34	15788.21	15770.33	23259.19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

Answer set Programming:

Reduce the design of protocols to the computation of answer sets

logic program ⇒ encoding of a problem
answer sets (model) ⇒ solutions of a problem



Specification: Behavioural Properties

Time dependant behavioural properties of Concurrent Programs:

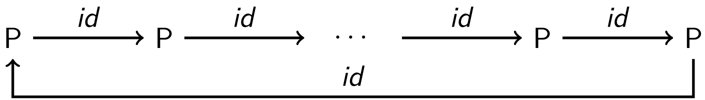
- safety
- liveness

Specified in a Temporal Logic, i.e., Computation Tree Logic (CTL):

- path quantifiers: for all paths **A**, for some paths **E**
- temporal operators: eventually **F**, globally **G**, next **X**,....

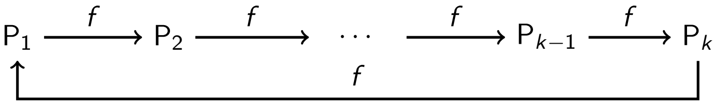
Process structure: encoded as a function f

either the identity function id



(Dijkstra's semaphore)

or a generator of a cyclic group $\{id, f, \ldots, f^{k-1}\}$ of order k



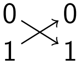
(Peterson's algorithm)

A 2-process protocol

Given the specification φ

Behavioural property: $AG\ \neg(\mathtt{x}_1 = \mathtt{u} \wedge \mathtt{x}_2 = \mathtt{u})$ (mutual exclusion)

Structural property:



(A) encode it as ASP Program P

```
false :- not ag(neg(and(local(p1,u),local(p2,u)))).
```

(B) compute the answer sets of P

(C) decode the protocols from the answer sets

$x_1 := t; x_2 := t; y := 0$

P_1 : $\mathit{true} \rightarrow$ if
$\quad \mathtt{x}_1 = \mathtt{t} \wedge \mathtt{y} = 0 \rightarrow \mathtt{x}_1 := \mathtt{u};\ \mathtt{y} := 0;$
$\| \quad \mathtt{x}_1 = \mathtt{t} \wedge \mathtt{y} = 0 \rightarrow \mathtt{x}_1 := \mathtt{w};\ \mathtt{y} := 1;$
fi

P_2 : $\mathit{true} \rightarrow$ if
$\quad \mathtt{x}_2 = \mathtt{u} \wedge \mathtt{y} = 1 \rightarrow \mathtt{x}_2 := \mathtt{t};\ \mathtt{y} := 1;$
$\| \quad \mathtt{x}_2 = \mathtt{t} \wedge \mathtt{y} = 1 \rightarrow \mathtt{x}_2 := \mathtt{w};\ \mathtt{y} := 0;$
fi

Complexity of the synthesis procedure

Theorem

For any number $k > 1$ of processes, for any symmetric program structure σ over $\mathcal{L}$ and $\mathcal{D}$, and for any CTL formula φ, an answer set of the logic program $\Pi_\varphi \cup \Pi_\sigma$ can be computed in

- *(i) exponential time w.r.t. k,*
- *(ii) linear time w.r.t. $|\varphi|$, and*
- *(iii) nondeterministic polynomial time w.r.t. $|\mathcal{L}|$ and w.r.t. $|\mathcal{D}|$.*

Experimental results

Specification:

- Mutual Exclusion (ME)
- Starvation Freedom (SF)
- Bounded Overtaking (BO)
- Maximal Reactivity (MR)

Synthesized k-process concurrent programs:

Program	Satisfied Properties	$\|ans(P)\|$	Time (sec)
mutex for 2 processes	*ME*	10	0.011
	ME	10	0.012
	ME, *SF*	2	0.032
	ME, *SF*, *BO*	2	0.045
	ME, *SF*, *BO*, *MR*	2	0.139
mutex for 3 processes	*ME*	9	0.036
	ME	14	0.036
	ME, *SF*	6	3.487
	ME, *SF*, *BO*	4	4.323

Conclusions

Verification Framework, which is parametric with respect to

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

Instantiated to prove partial correctness of integer and array C programs

Implemented and available as a stand-alone system: the **VeriMAP** tool, which is competitive with respect to others CLP-based software model checkers.

Synthesis Framework, a fully declarative solution

- reduces the design of a concurrent program to the design of its formal specification
- independent of the ASP solver