Program Transformation and Its Applications to Software Synthesis and Verification

Maurizio Proietti

Joint work with:

Fabio Fioravanti (Univ. D'Annunzio, Pescara, Italy),

Alberto Pettorossi (Univ. Tor Vergata, Rome, Italy),

Valerio Senni (Univ. Tor Vergata, Rome, Italy)

IASI – 1 Ottobre 2010

Correctness of Software

Safety and business-critical applications need dependable software.

Traditional validation and testing methodologies are not always adequate: they do not guarantee that software artifacts meet their specifications in all cases.

Logic-based methods aim at mechanically proving the correctness of software wrt formal specifications.

Overview

- Program Verification: proof of program properties

- Program Synthesis: automatic derivation of programs from first order logic specifications

- Program Transformation: automatic improvement of programs

A Bit of History

Automated Theorem Proving

Leibniz [1666] Calculus ratiocinator, Lingua characteristica universalis
Frege [1879] First Order Logic
Hilbert's program [early '900] Formalization of mathematics,
prove the consistency of Arithmetics by finitist methods,
the decision problem for FOL
Presburger [1929] Decision procedure for the FO theory of addition

Gödel [1931], Church-Turing [1936-7] Undecidability of Arithmetics and FOL

Decidable theories

Tarski [1951] First order theory of real numbers is decidable
Rabin [1969] Monadic Second Order Logic
Description logics [1990's] Ontologies, Semantic Web

General methods (based on application strategies)

Robinson [1965] Resolution
Kowalski [1974] Logic Programming
Jaffar-Lassez [1987] Constraint Logic Programming
CADE 2009: The Vampire resolution-based theorem prover solves 181/200 problems of the annual competition

A Bit of History

Verification

Turing [1936] Undecidability of the Halting Problem

Floyd [1967] Inductive assertions for flowchart programs
Hoare [1969] FOL axiomatization of the correctness of ALGOL programs
Pnueli [1977] Temporal logics for the verification of concurrent programs
Clarke-Emerson [1980] Model Checking

Synthesis

Waldinger [1969] Using resolution for synthesis of LISP programs
Clark-Hogger [1977-81] Synthesis of logic programs
Clarke-Emerson [1981] Synthesis of concurrent programs

Transformation

[1960's] Equivalence of flowchart schemas
Paterson-Hewitt [1970] Recursive schemas are more expressive than flowcharts
Burstall-Darlington [1977] Rule-based transformation of functional programs
[Hogger 1981,Tamaki-Sato 1984] Rule-based transformation of logic programs

Program Transformation

Rule-based Program Transformation

Initial program $\quad P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_n \quad$ Final program

where '→' is an application of a transformation rule.

- Program transformation separates the correctness and the efficiency concerns during program development.
- P_0 can easily be proved correct wrt a given specification and semantics M.
- Each rule application preserves the semantics:
  $M(P_0) = M(P_1) = \cdots = M(P_n)$
- The application of the rules is guided by a strategy which guarantees that P_n is more efficient than P_0.

An Example: Approximate Matching

Classical matching: S: L **P** R

Approximate matching:

Given two lists of integers $P=[x_1,\ldots,x_n]$ and S, and an integer K, match(P,S,K) iff there exists a subsequence $Q=[y_1,\ldots,y_n]$ of S s.t., for $i=1,\ldots,n$, $|x_i-y_i| \leq K$.

P: 2 0

S: 5 0 4 1 4 3 3 0 3 6 5 1 4

Q

max-diff(P,Q) $\leq$ 2

Constraint logic program for approximate matching:

S = L :: Q :: R

Match:

```
match(P,S,K) :- append(L,Q,A), append(A,R,S), max-diff(P,Q,K).
append([ ],S,S).
append([X|S],T,[X|U]) :- append(S,T,U).
max-diff([ ],[ ],K).
max-diff([X|S ],[Y|T],K) :- |X-Y|≤K, max-diff(S,T,K).
```

Approximate Matching (2)

- Suppose that we want to use the Match program for queries of the form:

  match([2, 0], S, 2)

- Add a new clause to Match:

  C: sp_match(S) :- match([2, 0], S, 2).

- $\{C\} \cup$ Match has a generate and test behaviour: subsequences Q of S are generated and then the test diff([2,0],Q,2) is performed.

- Derive an efficient program by applying a sequence of transformation rules according to a transformation strategy:

$$\{C\} \cup \text{Match} \;\rightarrow\; \cdots \;\rightarrow\; \text{Sp\_Match}$$

Specialized Approximate Matching

Sp_Match:

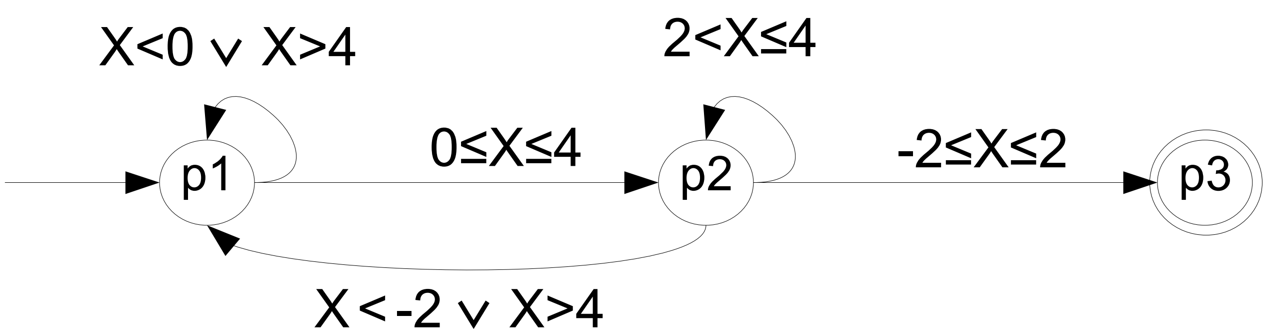
sp_match(S) :- p1(S).

p1([X|S]) :- 0≤X≤4, p2(S).
p1([X|S]) :- (X<0 ∨ X>4), p1(S).

p2([X|S]) :- -2≤X≤2, p3(S).
p2([X|S]) :- 2<X≤4, p2(S).
p2([X|S]) :- (X < -2 ∨ X>4), p1(S).

p3(S).

Sp_Match has a O(|S|) running time for an input sequence S and corresponds to a deterministic finite automaton



Correctness of the Transformation Rules

- The transformation rules (e.g., unfolding, folding, constraint replacement, clause replacement) replace a set of clauses by an equivalent one.

$$M(P_0) \models \boxed{q} \leftrightarrow \boxed{r}$$

P_0: $p \text{ :- } \boxed{q.}$ / $q \text{ :- } r.$ / $r.$ $\longrightarrow$ P_1: $p \text{ :- } \boxed{r.}$ / $q.$ / $r.$

$M(P_0) = \{p,q,r\}$ $=$ $M(P_1) = \{p,q,r\}$

- In general, replacement does not preserve the least model semantics:

$$M(P_0) \models \boxed{q} \leftrightarrow \boxed{p}$$

P_0: $p \text{ :- } \boxed{q.}$ / $q \text{ :- } r.$ / $r.$ $\longrightarrow$ P_1: $p \text{ :- } \boxed{p.}$ / $q.$ / $r.$

$M(P_0) = \{p,q,r\}$ $\neq$ $M(P_1) = \{q,r\}$

Correctness of theTransformation Rules

Replacement of equivalent formulas is partially correct (or sound):

$$M(P_0) \supseteq M(P_1) \supseteq \cdots \supseteq M(P_n)$$

if (i) P is a definite program (no negative literals in the premises)

(ii) M(P) is the least model of P

Correctness Issues

- Sufficient conditions for total correctness:

$$M(P_0) = M(P_1) = \cdots = M(P_n)$$

- General programs (negative literals in the premises)
- Various semantics: least model, perfect model, stable model, ...

Transformation Strategies

Transformation strategies are directed by syntactical features of programs

- Avoding multiple visits of data structures and repeated computations by eliminating multiple occurrences of variables from bodies of clauses
- Avoiding the computation of unnecessary values by eliminating existential variables (variables occurring in the body and not in the head)
- Reducing nondeterminism by avoiding multiple clauses for the same predicate definition
- Specializing programs to the context of use by pre-computing partially instantiated literals

Approximate String Matching Revisited

Initial program {C} ∪ Match:

sp_match(S) :- match([2, 0], S, 2).   Partially instantiated literal

match(P,S,K) :- append(L,Q,A), append(A,R,S), diff(P,Q,K).

Existential and multiple occurrences of list variables

Final program Sp_Match:

sp_match(S) :- p1(S).

p1([X|S]) :- 0≤X≤4, p2(S).
p1([X|S]) :- (X<0 ∨ X>4), p1(S).

p2([X|S]) :- -2≤X≤2, p3(S).
p2([X|S]) :- 2<X≤4, p2(S).
p2([X|S]) :- (X<-2 ∨ X>4), p1(S).

p3(S).

No multiple occurrence of list variables.
No existential variables.
Clauses are mutually exclusive.

Power of Strategies

Transformation strategies face undecidability limitations, e.g., the problem of checking whether or not from a given program we can derive a program without existential variables is undecidable.

Issues about Strategies

- In general, transformation strategies are based on heuristics and are evaluated in an experimental way
- For specific classes of programs the transformation strategies can be proved successful in terms of transformation times and speed-up.

Experimental Evaluation of Strategies

- Many transformation rules and strategies are implemented in the MAP system: http://www.iasi.cnr.it/~proietti/system.html
- Experimental results on matching and parsing problems

Program	Query	Transformation Time (s)	Speedup
String Matching	match([aab],S)	0.07	6.8×10^3
Multi Matching	mmatch([[aaa],[aab]],S,N)	0.28	6.8×10^3
Reg.Expr. Matching	re_match(aa*b,S)	0.21	3.0×10^6
Context Free Parsing	parse(g,[s],W)	1.62	87.1
Approximate Matching	match([2,0,4], S, 2)	1.89	46
Approx. Multi Matching	mmatch([[1,1],[1,2]], S, 1)	2.11	45

Program Synthesis by Transformation

The Transformational Synthesis Method

Program Synthesis: Given a logic program P and a first order formula $\varphi[X]$, derive a logic program Q defining a predicate r(X) such that, for all ground terms t:

$$M(P) \models \varphi[t] \quad \text{iff} \quad M(Q) \models r(t)$$

Maximum of a nonempty list:

P: member(X,[Y|L]) :- X=Y.

member(X,[Y|L]) :- member(X,L).

$\varphi[L,M]$: $member(M,L) \wedge \forall X\, (\, member(X,L) \rightarrow X \leq M)$

Clause form of $\varphi[L,M]$:

CF: r(L,M) :- member(M,L), $\neg$ greater(L,M).

greater(L,M) :- member(X,L), X > M.

Perfect model

$P \cup CF$ is correct: For all L, M, $M(P) \models \varphi[L,M]$ iff $M(P \cup CF) \models r(L,M)$

… but inefficient: generate an element M in L and test M is an upper bound [$O(|L|^2)$]

The Transformational Synthesis Method (2)

Derive an efficient program by (i) eliminating multiple and existential variables and
(ii) eliminating negation

CF: r(L,M) :- member(M,L), ¬ greater(L,M).
greater(L,M) :- member(X,L), X > M.

Elimination of multiple and existential variables
Elimination of negation

Q: r([X|L],M) :- s(X,L,M).
s(X,[ ],M) :- M=A.
s(X,[Y|L],M) :- Y ≤ X, s(X,L,M).
s(X,[Y|L],M) :- X ≤ Y, s(Y,L,M).

Q is correct: For all L, M, $M(P) \models \varphi[L,M]$ iff $M(Q) \models r(L,M)$

… and efficient: while visiting the list, keep the maximum so far [O(| L |)]

Issues in Transformational Synthesis

- Find suitable synthesis strategies based on heuristics (e.g., by composing several transformation strategies)

- Find specific classes of programs where the synthesis strategies can be proved successful in terms of synthesis times and speed-up.

Power of Transformational Synthesis

- Weak monadic second order theory of 1 successor (WS1S) [Buchi '60]

  $n ::= N \mid 0 \mid succ(n)$
  $\varphi ::= n_1 > n_2 \mid n_1 = n_2 \mid n \in S \mid S_1 = S_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists N\ \varphi \mid \exists S\ \varphi$

  where N is a variable ranging over natural numbers and S is a variable ranging over finite sets of natural numbers.

- WS1S is decidable in $\left.2^{\cdot^{\cdot^{\cdot^{2^{n}}}}}\right\}dn$ time complexity, for some $d>0$.

- For every WS1S formula the transformational method synthesizes a program with linear time complexity.

- The transformation strategy has $\left.2^{\cdot^{\cdot^{\cdot^{2^{n}}}}}\right\}dn$ worst case time complexity.

Verification of Program Properties by Program Transformation

Theorem Proving by ProgramTransformation

Given a program P and a closed first order formula φ, check whether or not

$$M(P) \models \varphi$$

The transformational proof method:

$$\begin{array}{lcl} \text{Closed formula} & \varphi & \\ & \downarrow & \text{clause form} \\ & CF & \\ & \downarrow & \text{elimination of existential variables} \\ \text{Propositional program defining a predicate } r & Q & \end{array}$$

such that: $M(P) \models \varphi$ iff $M(Q) \models r$

$M(Q) \models r$ is decidable in $O(|Q|)$ time

The Transformational Proof Method

Given a program

P: `member(X,[Y|L]) :- X=Y.`
   `member(X,[Y|L]) :- member(X,L).`

and a closed first order formula ("every list of numbers has an upper bound")

φ: $\forall L\ (list(L) \rightarrow \exists U\ \forall X\ (\ member(X,L) \rightarrow X \leq U\ ))$

we want to prove:

$M(P) \models \varphi$

Step 1. Clause-Form Transformation

$$\varphi: \qquad \forall L\ (list(L) \rightarrow \exists U\ \forall X\ (\ member(X,L) \rightarrow X \leq U\ ))$$

$$r \equiv \neg \underbrace{\exists L(list(L) \wedge \neg \underbrace{\exists U \neg \underbrace{\exists X\ (\ member(X,L) \wedge \neg X \leq U}_{c}\ )}_{b})}_{a}$$

Clause-Form:

CF: r :- ¬ a.

a :- list(L), ¬ b(L).

b(L) :- list(L), ¬ c(L,U).

c(L,U) :- X > U, list(L), member(X,L).

Program with existential variables

$$M(P) \models \varphi \qquad \text{iff} \qquad M(P \cup CF) \models r$$

Step 2. Elimination of Existential Variables

The strategy for the elimination of existential variables returns:

Q: r :- $\neg$ a.

a :- d.

d :- d.

s.t. $M(P) \models \varphi$ iff $M(Q) \models r$

Step 3. Computation of the Perfect Model

Q: r :- ¬ a. S2: stratum 2

a :- d.

d :- d. S1: stratum 1

1. Compute the least model of S1:

$\{\,\} \xrightarrow{T_{S1}} \{\,\}$ where T_{S1} is the immediate consequence operator (= one-step deduction)

{ } is the least fixpoint of T_{S1}, hence M(S1) = { }

2. Transform S2 using M(S1):

r.

M(Q) = (M({r.}) ∪ M(S1)) = {r}

$\Rightarrow \quad M(Q) \models r \quad \Rightarrow \quad M(P) \models \varphi$

Power of the Proof Method

The transformational proof method is a decision procedure for WS1S.

Experimental evaluation

Examples run by the MAP transformation system
(www.iasi.cnr.it/~proietti/system.html)

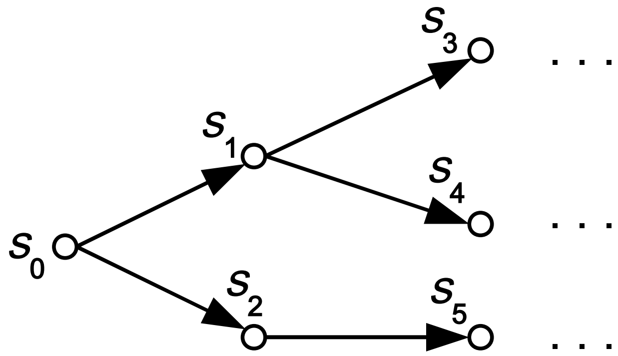
Constraints are handled using the clp(R) module of SICStus Prolog
(implementing a variant of Fourier-Motzkin variable elimination)

Property	Time (PM 1.73)
$\forall L\ \exists U\ \forall Y\ (\ member(Y,L) \rightarrow Y \leq U\ )$	31 ms
$\forall L\ \forall Y\ (\ (sumlist(L,Y) \wedge Y > 0) \rightarrow \exists X\ (member(X,L) \rightarrow X > 0)\ )$	15 ms
$\forall L\ \forall M\ \forall N\ (\ (ord(L) \wedge ord(M) \wedge sumzip(L,M,N)) \rightarrow ord(N)\ )$	16 ms
$\forall L\ \forall M\ \forall X\ \forall Y\ (\ (leqlist(L,M) \wedge sumlist(L,X) \wedge sumlist(M,Y)) \rightarrow X \leq Y\ )$	16 ms

Model Checking Infinite State Systems by Program Transformation

Verification of Infinite State Systems

- The behaviour of a concurrent system can represented as a state transition system which generates infinite computation paths:



- Properties are expressed in the Temporal Logic CTL, a propositional logic augmented with:
  (1) quantifiers over paths: E (Exists), A (All), and
  (2) temporal operators along paths: F (Future, there exists a state in the path),
  G (Globally, for all states of the path).

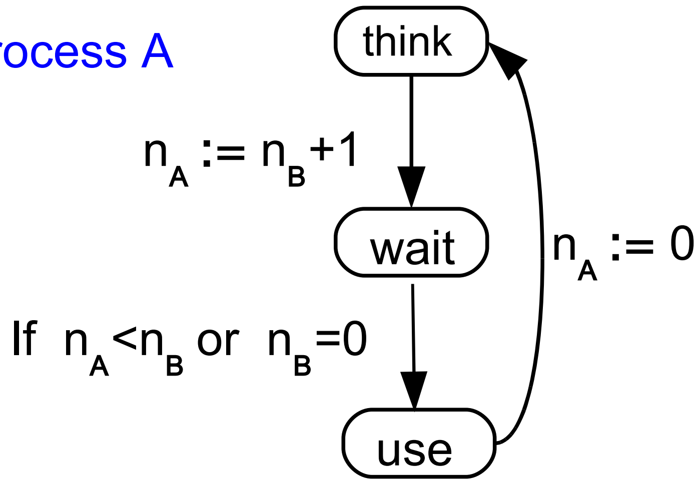
- If the set of states is finite, then CTL is decidable in polynomial time.

- CTL is undecidable for: (1) infinite state systems (e.g., integer variables) and (2) parameterized systems (families of finite-state systems).

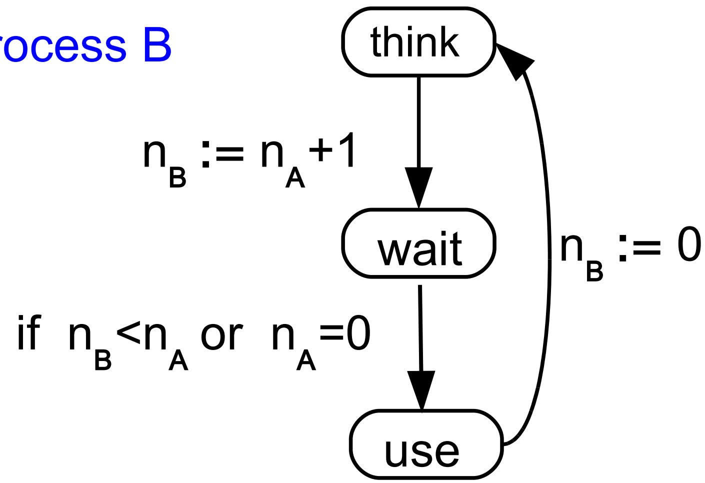
The Bakery Protocol (Lamport)

Each process has: control state: $s \in \{think, wait, use\}$ and counter: $n \in N$

Process A



Process B



System: A || B

$\langle think,0,think,0\rangle \rightarrow \langle wait,1,think,0\rangle \rightarrow \langle wait,1,wait,2\rangle \rightarrow \langle use,1,wait,2\rangle \;\; \ldots$

Mutual Exclusion: $\langle think,0,think,0\rangle \models \neg$ EF unsafe

where, for all n_A, n_B: $\langle use,n_A,use,n_B\rangle \models$ unsafe

Temporal Properties as Constraint Logic Programs

A system S and the temporal logic are encoded by a constraint logic program P_S:

- the transition relation is encoded by a binary predicate trans:

```
trans(<think,A,S,B>,<wait,A1,S,B>) :- A1=B+1.
trans(<wait,A,S,B>,<use,A,S,B>) :- A<B.
trans(<wait,A,S,B>,<use,A,S,B>) :- B=0.
trans(<wait,A,S,B>,<use,A1,S,B>) :- A1=0.
```

- the satisfaction relation |= is encoded by a binary predicate holds:

```
holds(<use,A,use,B>, unsafe).
holds(S, not(P)) :- ¬ holds(S, P).
holds(S, ef(P)) :- holds(S, P).
holds(S, ef(P)) :- trans(S,T), holds(T, ef(P)).
```

- the property to be verified is encoded by a predicate prop:

```
prop :- holds(<think,0,think,0>, not(ef(unsafe))).
```

Protocol Verification Via Program Transformation

- The encoding is correct:
  $\langle think,0,think,0 \rangle \models \neg$ EF unsafe iff $M(P_S) \models prop$

- Bottom-up construction of $M(P_S)$ from facts does not terminate because $M(P_S)$ is infinite. Top-down evaluation of P_S from prop does not terminate due to infinite computation paths.

- Transformation-based Verification Method:

  1) specialize P_S to the query prop:

  $$P_S \rightarrow \cdots \rightarrow Q \qquad \text{s.t.} \qquad M(P_S) \models prop \;\; \text{iff} \;\; M(Q) \models prop$$

  2) keep only the clauses dep(prop, Q) on which the predicate prop syntactically depends:

  $$prop \in M(Q) \;\; \text{iff} \;\; prop \in M(dep(prop, Q))$$

  3) construct bottom-up the model of dep(prop, Q).

Experimental Results Using the MAP System

Protocol	Property	Time (s)
Bakery (mutual exclusion)	safety: $\neg$EF unsafe	0.05
	liveness: AG(wait $\rightarrow$ AF use)	0.13
Ticket (mutual exclusion)	safety: $\neg$EF unsafe	0.04
	liveness: AG(wait $\rightarrow$ AF use)	0.10
Sleeping Barber	safety	0.03
Office Light Control	safety	0.10
Petri Net	safety	0.08
Berkeley RISC (cache coherence)	safety	0.07
Xerox Dragon (cache coherence)	safety	0.07
DEC Firefly (cache coherence)	safety	0.05
Illinois Univ. (cache coherence)	safety	0.06
MESI (cache coherence)	safety	0.07
MOESI (cache coherence)	safety	0.08
Synapse N+1 (cache coherence)	safety	0.04
IEEE Futurebus+ (cache coherence)	safety	0.22

Ongoing Work

- More expressive logics for path properties:
  - LTL / CTL* [PPS09]
  - ω-regular languages [PPS10]
- Proving properties of logic programs on infinite structures
  (some work already in [PPS10] for infinite lists)
- Synthesis of reactive systems (e.g., protocols)
- Modelling and verification of Business Processes

  [joint work with Missikoff, Smith]