Journal of Computer Science 6 (6): 641-647, 2010
ISSN 1549-3636

Specialization of Recursive Predicates from Positive Examples Only

Moussa Demba
Department of Computer and Information Sciences,
Aljouf University Skaka, Saudi Arabia

Abstract: Problem statement: Given an overly general (definite) program P and its intended semantics ϕ (the programmer's intentions) where P does not satisfy ϕ, find out a new version P' of P such that P' satisfies ϕ. **Approach:** We proposed an approach for correcting overly general programs from positive examples by exploiting program synthesis techniques. The synthesized program, P', is a specialization of the original one, P. In contrast to the previous approaches for logic program specialization, no negative examples were given as input but they will be discovered by the algorithm itself. The specialization process is performed according to the positive examples only. A method for refining logic programs into specialized version was then proposed. **Results:** The proposed approach was able to correct overly general programs using positive examples. We showed that positive examples can also be used for inducing finite-state machines, success sequences, that models the correct program. The failing sequences also exploited by theorem proved to produce counter-examples as in model checking, by composing substitutions used for inducing failing sequences. **Conclusion:** The contribution of the study was mainly the use of specification predicates to specialize an overly general logic program.

Key words: Program specialization, theorem proving, positive/negative examples, folding/unfolding rules, finite-state machine

INTRODUCTION

Our aim is to present a top-down approach for logic program specialization w.r.t the intended speciation which is a first-order formula of the following form: $\forall \bar{x} \exists \bar{Y} \phi(\bar{x}, \bar{Y}) = \forall \bar{x} \exists \bar{Y} \Gamma(\bar{x}, \bar{Y}) \leftarrow \Delta(\bar{x})$ (or $\Gamma \leftarrow \Delta$ for short) where Γ and Δ are conjunction of atoms. The problem we are interested in can be stated as follows:

Given: An overly general (definite) program P = ($E^+ \cup C$) where E^+ is a recursive sub-program defining positive examples, C is supposed to be the set of clauses defining the overly general predicates (i.e. the incorrect component of P) and the intended semantics ϕ for P (the programmer's intentions) with $M(P)| = \phi$ where M (P) denotes the least Herbrand model of P.

Find: A definite program D, called a specialization of C, such that $M(D) \subseteq M(C)$ and $M(P') \models \phi$ where P' = $(P/C) \cup D$.

The program P' is called a correct specialization of P' w.r.t E^+ if $M(P') \subseteq M(P)$, $M(E^+) \subseteq M(P')$ and for any negative example e−, $M(P') \not\models e-$.

Roughly the approach takes an overly general program P and its intended semantics ϕ and tries to produce a program P' that is guaranteed to satisfy the specification and therefore does not require verification. We outline a top-down method for synthesizing a correct and consistent logic program P' that satisfies the given specification. Moreover, the negative examples E^- correspond to ground atoms that are not deducible from P' and are automatically discovered during the specialization process.

For example, assume we are given the overly general program P = ($E^+ \cup C$) where:

$$E^+ \begin{cases} \text{even}(0) & \leftarrow \\ \text{even}(s^2(n)) & \leftarrow \text{even}(n) \end{cases}$$

and

$$C \begin{cases} \text{len}([],0) & \leftarrow \\ \text{len}([a \mid x], s(n)) & \leftarrow \text{len}(x,n) \end{cases}$$

and its intended specification:

$$\phi: \text{even}(n) \leftarrow \text{len}(x,n)$$

supposed to establish the claim "if n is the length of list x, then n is even".

For this specification P is false as we discover while attempting to prove it, for example there are particular values of the list x that generate negative examples: It is the case where the number of elements in x is odd and the negative examples even ($s^{2k+1}(0)$) k = 0,..,n will be generated (s is the successor function). But with the specialized version D of C, the new program P' = $(P\backslash C)\cup D$ satisfies ϕ up to renaming the predicate len by len2 that is defined as follows:

$$D+\begin{cases} \text{len}2\,([],0) & \leftarrow \\ \text{len}2[a,b\,|\,x],s^2(n)) \leftarrow \text{len}2(x,n) \end{cases}$$

The new predicate len2 is called a specialization predicate of ϕ w.r.t the predicate len. The proposed method consists to synthesize D.

Throughout the study, Γ, Δ and Λ denote conjunctions of atoms; ϕ denotes the intended specification (a first-order formula); A and B denote atoms and θ denotes substitution. In all formulas, existentially quantified variables are distinguished from universal variables by giving them upper-case letters. A program is a set of definite clauses, denoted by calligraphic letters: P, Q,....

MATERIALS AND METHODS

Let C be a conjunction of atoms. Then $\mu(C) = \varnothing$ if C is the predicate true and the multi-set of the atoms of C otherwise.

Definition 1: A conjunction of atoms C_1 is a specialization of (or is syntactically less general than) a conjunction of atoms C_2 (denoted $C_1 \preceq C_2$) with a substitution θ iff $\mu(C_2\theta) \subseteq \mu(C_1)$ (Flener and Deville, 1993).

For example, suppose C_1 = p(a, x)$\wedge$q (y), C_2 = p(v,w) and θ = {v/a, w/x} we have $C_1 \preceq C_2$. Indeed, $\mu(C_2\theta)$ = {p (a, x)} and $\mu(C_1)$ = {p (a, x),q(y)}.

The following definition expresses the relation of generality between two horn clauses.

Definition 2: A definite clause $A\leftarrow\Delta$ is represented by a couple of elements (A, Δ). A clause (A_1, Δ_1) is a specialization of a clause (A_2, Δ_2), denoted $(A_1, \Delta_1) \preceq (A_2, \Delta_2)$, with a substitution θ iff (i) $\mu(A_2\theta)\subseteq\mu(A_1)$ and (ii) $\mu(\Delta_2\theta) \subseteq \mu(\Delta_1)$.

For example for (A_1,Δ_1) = (r(a, x, y), {p(a, x) q(y)}) and (A_2, Δ_2) = (r(v,w, y), {p(v,w)}), we have $(A_1, \Delta_1) \preceq (A_2, \Delta_2)$ with the substitution θ = {v/a, w/x} as $\mu(A_2\theta)$ = {r(a, x, y)} and $\mu(\Delta_2\theta)$ = {p(a, x)}.

The following definition expresses the relation of generality between two logic programs.

Definition 3: Let P_1 and P_2 be two definite logic programs, $\{c_1,\ldots, c_n\}$ the set of clauses of P_1 and $\{d_1,\ldots, d_m\}$ the set of clauses of P_2. P_1 is a specialization of P_2 (denoted $P_1 \preceq P_2$) iff for all $1\leq i\leq n$, there exists $1\leq j\leq m$ s.t. $c_i \preceq d_j$.

Example: Let P_1 and P_2 be two definite programs:

$$P_1: \begin{cases} c_1\ p(s,(0),(0) & \leftarrow \\ c_2: p(s^2(x)\ s(y)) \leftarrow P(x,y),q(y) \end{cases}$$

$$P_2: \begin{cases} d_1: p(0,0) & \leftarrow \\ d_2: p(s\ (0),0) & \leftarrow \\ d_3: p(s^2(x),s(y)) \leftarrow p(x,y) \end{cases}$$

Then P_1 is a specialization of P_2 as $c_1 \preceq d_2$ and $c_2 \preceq d_3$.

Definition 4 (Specialization predicate): Let P_1 and P_2 be two definite programs defining the predicates p_1 and p_2 respectively. If P_1 is a correct specialization of P, i.e., $p_1 \preceq p_2$, w.r.t the intended specification ϕ, then p_1 is called a specialization predicate of p_2 w.r.t ϕ.

Proposition 1: Let ϕ: $\Gamma\leftarrow\Delta$, p_2 be the intended specification of the program P. If p_1 is a specialization predicate of ϕ with respect to the predicate p_2, then we have $M(P\cup P_1) \models ((\Gamma, \Delta, p_2)\leftarrow p1)$ where P_1 defines the predicate p_1.

Proposition 2: If p_1 is a specialization predicate of ϕ with respect to the predicate p_2, then the two formulas (1) and (2) are equivalent:

$$(\Gamma \leftarrow \Delta p_2) \leftarrow p_1 \tag{1}$$

$$\Gamma \leftarrow \Delta, P_1 \tag{2}$$

Proof: It is easy to see that the formula (1) is equivalent to:

$$\Gamma \leftarrow \Delta, p_2 p_1 \tag{3}$$

Moreover, if $p_1 \preceq p_2$, $(p_2 \leftarrow p_1)$ is a theorem Therefore, the formula (3) is equivalent to:

$$\Gamma \leftarrow \Delta, p_1 \qquad (4)$$

For example, the formula (even (n)←len(x,n), len_2(x, n)) is equivalent to even(n)←len2(x, n).

Notation 1: Hereafter and for the sake of simplicity, the notation $<\phi|p>$ stands for $<\phi\leftarrow p>$.

In the definitions 5-8, we define the semantic calculus that allows given P and its intended specification ϕ such that P is faulty w.r.t ϕ, to find P' such that P' satisfies ϕ.

Transformation rules: The algorithm applies the transformation rules unfolding, folding and simplification (Sakurai and Motoda, 1988). Intuitively, unfolding is an extension of SLD-resolution and folding applies the induction hypotheses. Indeed, whereas an unfold step replaces a term that "matches" the conclusion of a definition in the program by the corresponding hypothesis, a folding step replaces a conjunction of atoms that match the hypothesis of an induction hypothesis by the corresponding conclusion.

There are two kinds of unfolding rules: The negation as failure inference (nfi for short) that replaces a predicate call, at the right hand side, by the corresponding body and the definite clause inference (dci for short) that replaces a predicate call, at the left hand side, by the corresponding body (Sakurai and Motoda, 1988).

To specialize the original program, it is vital to keep trace of substitutions in the specialization predicates, denoted IO (for Input Output). Therefore each transformation rule is associated with a procedure construction of the corresponding specialization predicates. The application of an unfolding rule on a formula ϕ_0 generates a finite set of formulas ϕ_i, i = 1,...,k, such that ϕ_0 follows from the ϕ_i's in the least Herbrand model of the program under consideration. Each formula ϕ_i is associated with a specialization predicate, as it can be an overly general clause and defined by a program, noted Q_R where R is the applied rule. If ϕ_i is trivially true, its associated specialization predicate, IO_i, is set to true. If ϕ_i is trivially false (covers only the negative examples), then its associated specialization predicate is set to false and in this case all clauses containing this predicate will not be included in the synthesized program D. The process is iterated until all the formulas newly generated are trivial. The arguments of the input output predicate IO_i are those that appear in the corresponding formula ϕ_i.

The folding rule (cutr for short) is necessary for synthesizing recursive predicates.

Definition 5 (negation as failure inference): Let P be a program, ϕ_0: $\Gamma \leftarrow \Delta$, A a formula and C = $\{c_1,...,c_k\}$ the set of clauses of P such that $c_i : B_i \leftarrow \Delta_i$ and suppose there is a substitution θ_i s.t $B_i\theta_i = A\theta_i$. Then the application of the rule of nfi on ϕ_0 w.r.t to the atom A yields a conjunction of k formulas:

$$\frac{<\phi_0 : (\Gamma \leftarrow \Delta, A) \mid IO_0>}{<\phi_i : (\Gamma \leftarrow \Delta, \Delta_i)\theta_i \mid IO_i>_{i=1,\ldots,k}}(\text{nfi})$$

where, IO_i is the specialization predicate of ϕ_i w.r.t the predicate A. Hence $Q_{nfi} = \{IO_0\theta_i \leftarrow IO_i\}_{i=1,\ldots,k}$.

Definition 6: (definite clause inference) Let P be a program, ϕ_0: $\Gamma \leftarrow \Delta$ a formula and C = $\{c_1,..., c_k\}$ the set of clauses of P such that c_i: $B_i \leftarrow \Delta_i$ and suppose there is a substitution θ_i s.t $B_i\theta_i = A\theta_i$. Then the application of the rule of nfi on ϕ_0 w.r.t to the atom A yields a disjunction of k formulas:

$$\frac{<\phi_0 : (\Gamma, A \leftarrow \Delta) \mid IO_0>}{\mathrm{V}_{i=1,\ldots,k} <\phi_i : (\Gamma \leftarrow \Delta_i)\theta_i \leftarrow \Delta \mid IO_i>}(\text{dci})$$

where, IO_i is the specialization predicate of ϕ_i w.r.t the predicate A and $Q_{dci} = \{IO_0\theta_i \leftarrow IO_i\}_{i=1,\ldots,k}$. Unlike in Q_{nfi}, the substitution θ_i is an existential one (θ_i substitutes only existential variables of A) in Q_{dci}.

Definition 7 (folding rule or cutr): Let ϕ_0: $\Lambda \leftarrow \Pi$ and θ_1: $\Gamma \leftarrow \Delta_1$, Δ_2 be two formulas satisfying the following conditions: (i) ϕ_1 is obtained (directly or indirectly) from ϕ_0 by the rule of nfi, (ii) Δ_1 is an instance of Π, i.e., there is a substitution θ such that $\Pi\theta = \Delta_1$, (iii) for any local variable x in Π, $x\theta$ is a variable and does not occur other than in Δ_1 and (iii) θ replaces different local variables of Π with different local variables of Δ_1. Then replace ϕ_1 by ϕ_2:

$$\frac{\begin{array}{c}<\phi_0 : (\Lambda \leftarrow \prod) \mid IO_0> \\ <\phi_1 : (\Gamma \leftarrow \Delta_1, \Delta_2) \mid IO_1>\end{array}}{<\phi_2 : (\Gamma \leftarrow \Lambda\theta, \Delta_2) \mid IO_2>}(\text{cutr})$$

in this case $Q_{cutr} = \{IO_1 \leftarrow IO_0\theta, IO_2\}$ defines the predicate IO_1 in terms of IO_0 and IO_2 where IO_0, IO_1

and IO_2 are the associated specification predicates of ϕ_0, ϕ_1 and ϕ_2 respectively. ϕ_0 thus plays the role of the induction hypothesis. This important rule allows us to synthesize recursive predicates.

Definition 8 (Simpification or simp): The rule of simplification (simp) simplifies the atoms A and B if there is an existential substitution θ s.t Aθ = B:

$$\frac{<\phi:(A,\Gamma \leftarrow B,\Delta)|IO\ >}{<\phi'\ :(\Gamma\theta \leftarrow \Delta)|IO'>}(simp)$$

and

$$(simp)Q_{simp} = \{IO\theta \leftarrow IO'\}$$

All the rules are partially correct (Demba *et al*., 2005). Substitutions of variables during the specialization process are stored into the specialization predicates.

RESULTS

Let P be a logic program and ϕ its intended specification (intended to be true). If $M(P) \not\models \phi$, then P is buggy. Our goal is (i) to isolate the set of incorrect axioms, denoted by C, of P w.r.t ϕ and (ii) to synthesize a sub-program, denoted by D, from the proof attempt of P w.r.t. to ϕ and (iii) to determine the program P' = $(P\backslash C)\cup D$ where D is a specialization of C and $M(P') \models \phi$.

Suppose P = $(E^+ \cup C)$ and D the synthesized program during the proof attempt of P w.r.t ϕ. Suppose C = $\{c_1,...,c_n\}$ and D = $\{d_1, …, d_m\}$, here is the specialization algorithm Fig. 1.

Hereafter, we will add clause numbers to incorporate into Fig. 2 and 3 of clause sequences in order to clarify which clauses have been resolved with.

Example 1: Consider the program P = $(E^+ \cup C)$ where C expresses that any natural number is odd:

$$c_5 : even(s\ (0)) \leftarrow \qquad |(IO_1()$$
$$c_6 : even(s^2(n)) \leftarrow ood(n))\ |(IO_2(n)$$

together with the intended specification

$$\phi : even(s(n)) \leftarrow odd(n)$$

It is clear that $M(P) \not\models \phi$, for example we have even(0) and odd(0). Therefore, the program P and specially the sub-program C, covers negative examples.

```
i← 1
While i≤ m do
If there exists a clause e ∈ C such that d_i ⪯ e then
i←i+1
else Exit
End while
If i > m then
D is a specialization of C
End If
```

Fig. 1: Program specialization algorithm

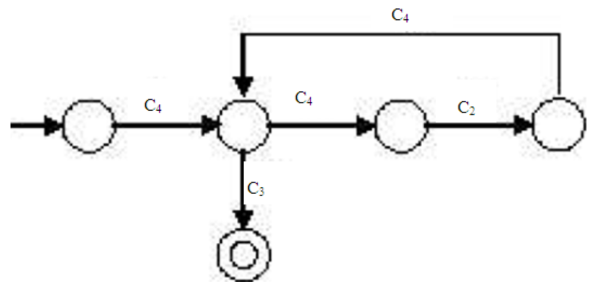

Fig. 2: Specialization of odd (n) w.r.t E+

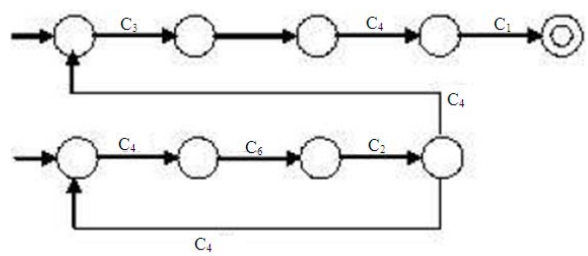

Fig. 3: Specialization of plus(x, y, Z) w.r.t E^+

To fix this problem, we need to specialize the predicate odd. To do that, assume IO_0 is a specialization predicate of odd associated toϕ:

$$\phi: even\ (s(n)) \leftarrow odd(n)\ |\ IO_0(n)$$

Note that we need to synthesize a definite program D defining the predicate IO_0 such that $M(E^+ \cup D) \models \phi$ with $\phi = even(s(n)) \leftarrow IO_0(n)$ and $M(D) \subseteq M(C)$. D is initially empty.

The specialization process of C w.r.t E^+ is in the way depicted in Fig. 2. The first step consists to unfold Á upon the atom odd (n) using the rule of nfi to obtain the following resultants:

$$c_5 : even(s\ (0))\ \leftarrow \qquad |(IO_1()$$
$$c_6 : even(s^2(n))\ \leftarrow ood(n))\ \ |(IO_2(n)$$

and IO_0 is defined in terms of IO_1 and IO_2 in Q_{nfi} as follows:

$$
\begin{array}{ll}
IO_0(0) & \leftarrow IO_1() \\
IO_0(s(n)) & \leftarrow IO_2(n)
\end{array}
$$

The clause c_5 corresponds to a negative example as $M(E^+) \models$ even (s(0)), then its associated specialization predicate IO_1 is set to false and the clause $IO_0(0) \leftarrow IO_1()$ is not included in D:

$$D = \varnothing \cup \{IO_0(s(n)) \leftarrow IO_2(n)\}$$

We apply again the rule of nfi on c_6 upon odd(n) to get:

$$
\begin{array}{llll}
c_7: & even(s^2(n)) & \leftarrow & | IO_3() \\
c_8: & even(s^3(n)) & \leftarrow odd(n) & | \ | IO_4(n)
\end{array}
$$

The clause c_7 corresponds to a positive example, then IO_3 is set to true and:

$$
\begin{array}{lll}
D = \{ & IO_0(s(n)) & \leftarrow IO_2(n) \\
& IO_2(0) & \leftarrow \\
& IO_2(s(n)) & \leftarrow IO_4(n)\}
\end{array}
$$

Next, we unfold c_8 using the rule of dci w.r.t c_2, to obtain:

$$c_9: even(s(n)) \leftarrow odd(n) \quad | IO_5(n)$$

and

$$
D = \begin{cases}
IO_0(s(n)) & \leftarrow IO_2(n) \\
IO_2(0) & \leftarrow \\
IO_2(s(n)) & \leftarrow IO_4(n) \\
IO_4(n) & \leftarrow IO_5(n)\}
\end{cases}
$$

as c_9 is an instance of ϕ, to complete the proof we can apply the folding rule (cutr), to obtain:

$$c_{10}: even(s(n)) \leftarrow even(s(n)) \quad | IO_6(n)$$

and the clause $IO_5(n) \leftarrow IO_0(n)$; $IO_6(n)$ is generated. The formula c_{10} can be simplified to true, IO_6 is then set to true. The final program D is then:

$$
\begin{array}{lll}
D = \{ & IO_0(s(n)) & \leftarrow IO_2(n) \\
& IO_2(0) & \leftarrow \\
& IO_2(s(n)) & \leftarrow IO_4(n) \\
& IO_4(n) & \leftarrow IO_5(n) \\
& IO_5(n) & \leftarrow IO_0(n)\}
\end{array}
$$

By eliminating the intermediate predicates a la Tamaki and Sato (1984), we get the final version:

$$
D \begin{cases}
IO_0(s(0)) & \leftarrow \\
IO_0 s^2(n)) | & \leftarrow IO_0(n)
\end{cases}
$$

and we have the correct specialization program P' = $(P \backslash C) \cup D$ of P w.r.t E^+ and $M(P') \models \phi$ where ϕ: $even(s(n)) \leftarrow IO_0(n)$ (according to the Proposition 2).

Note that the clause c_3 is automatically removed and c_4 is refined by specialization. The success sequences of clauses that cover only the positive examples, E^+, are of the following form $c_4 (c_4c_2c_4)^*c_3$ represented by the Fig. 2. Any other combination of clauses will cover negative examples, then leads to failure. From E^+, we can induce the finite-state machine of Fig. 2 that corresponds to the sub-program D.

The Fig. 2 can be interpreted as follows: the transition c_4 corresponds to the application of the rule of nfi while the transition c_2 corresponds to the application of the rule of dci. The loop means that recursive specialization is necessary, that is the application of the folding rule (cutr) is needed to complete the process.

The approach can also be applied to more complex specification. For example a specification with existential variables or an original program where the positive examples consist of different predicates as in the following example.

Example 2: Consider the specification:

$$\phi: plus(x, y, Z), sup(Z, x) \leftarrow nat(x), nat(y)$$

where the predicates sup, nat and plus are initially defined as usual by the program, say $P = (E^+ \cup C)$:

$$
E^+ \begin{cases}
c_1: sup(s,(x),0) & \leftarrow \\
c_2: sup(s,(x),s(y)) & \leftarrow sup(x,y) \\
c_3: nat(0) & \leftarrow \\
c_4: nat(s(x)) & \leftarrow nat(x)
\end{cases}
$$

$$C\begin{cases} c_5 : \text{plus}(0, x, x) & \leftarrow \\ c_6 : \text{plus}(s(x), y, s(z)) & \leftarrow \text{plus}(x, y, z) \end{cases}$$

sup(x,y) means that x>y, plus(x,y,z) means that z = x+y and nat(x) is true if x is a natural number. P does not satisfy its intended semantics ϕ for x = 0 and y = 0. Again, to fix the problem the predicate plus has to be specialized to IO_0 that is defined as follows:

$$D\begin{cases} IO_0(0s,(x),s(x)) & \leftarrow \\ IO_0(s,(x),y,s(z)) & \leftarrow IO_0(x,y,z) \end{cases}$$

Surprisingly, D is a specialization of C and $M(K \cup D) \models (\text{sup}(z, x) \leftarrow IO_0(x, y, z))$. Comparing C and D, we can say that the error was in the first clause of C, i.e., the underlined arguments. The correct program P' is then:

$$\varepsilon + \begin{cases} \text{sup}(s,(x),0) & \leftarrow \\ \text{sup}(s(x),s(y)) & \leftarrow \text{sup}(x,y) \\ \text{nat}(0) & \leftarrow \\ \text{nat}(s(x)) & \leftarrow \text{nat}(x) \end{cases}$$

$$D = \begin{cases} IO_0\big(0,s(x),s(x)\big) & \leftarrow \\ IO_0\big(s(x),y,s(z)\big) & \leftarrow IO_0\big(x,y,z\big) \end{cases}$$

The success sequences of clauses that cover only the positive examples, E^+, are depicted by the Fig. 3. Any other combination of clauses will cover negative examples, then leads to failure. From E^+, we can induce the finite-state machine of Fig. 3 that corresponds to the sub-program D.

From E^+, we can induce the finite-state machine of Fig. 3. The transitions c_3 and c_4 correspond Fig. 3. Specialization of plus(x,y,Z) w.r.t E^+ to the application of the rule of nfi and the transitions c_5 and c_6 correspond to the application of the rule dci. Again to complete the process, the application of the folding rule is needed.

DISCUSSION

Bostrom and Idestam-Alquist (1994; 1999) presented top down approaches such as the divide-and-conquer, covering and SPECTRE algorithms for logic program specialization using unfolding and clause deletion rules. One of the limitations of those algorithms is that the divide-and-conquer algorithm does not work when specializing clauses that define recursive predicates and the SPECTRE algorithm cannot synthesize recursive specifications. A bottom-up approach has been proposed in (Kanamori and Seki, 1986; Ferri *et al*., 2001; Leuschel and Massart, 2003). Ballis (2005) claimed that his approach can be applied as a top down or a bottom up approach. All those approaches are driven by (a finite set) positive and negative examples. It is not also clear how they handle cases when some positive examples are not included in the specification. Other works have been proposed to correct faulty specification (Protzen, 1996 Monroy, 2000) and all deal with faulty universally quantified equations.

To guarantee that all positive examples are included in the original program, we have proposed to represent them not as a set of ground terms but a recursive program denoted E^+. The intended specification we consider is not limited to Horn clauses but a first-order formula with universal and existential variables. The negative examples are not given as input but discovered during the proof process. Recursive predicates are synthesized, if needed.

CONCLUSION

We have presented a new way to specialize logic programs from positive examples only. With this approach recursive predicates can be obtained. We have shown that positive examples can be used for inducing finite-state machines (success sequences). The failing sequences could also be exploited by theorem proves to produce counter-examples as in model checking, by composing substitutions used for inducing failing sequences. The presented approach is implemented in Ocaml and integrated into the interactive proof assistant SPES (Demba *et al*., 2005). The contribution of the study is mainly the use of specification predicates to specialize an overly general logic program.

The framework presented here has two major advantages: (i) The positive examples defined in E^+ are guaranteed to be included both in the meaning of the original program and of the specialized version. Note that in (Ballis, 2005; Alpuente *et al*., 2001; Bostrom and Idestam-Almquist, 1999), E^+ consists of a finite set of ground atoms and it is not clear how they handle cases when some positive examples are not included in the original program. (ii) The specialization process is performed according to the positive examples only, no need to negative examples. (iii) It supports reasoning about specifications whose stat-spaces may be infinite.

But more works are needed to guarantee the termination of the procedure. This problem is due by the fact that the procedure is based on theorem proving techniques.

REFERENCES

Alpuente, M., F.J. Correa and M. Falaschi, 2001. Declarative debugging of functional logic programs. Elect. Notes Theor. Comput. Sci., 57: 17-40. DOI: 10.1016/S1571-0661(04)00266-X

Ballis, D., 2005. Rule based software verification and correction. Ph.D. Thesis, Universidad Politecnica de Valencia, Spain. http://hdl.handle.net/10251/1948

Bostrom, H. and P. Idestam-Alquist, 1994. Specialization of logic programs by pruning SLD-tree. Proceeding of the 4th International Workshop on Inductive Logic Programming, Sept. 1994, Bonn Germany, pp: 12-14. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3200

Bostrom, H. and P. Idestam-Almquist, 1991. Induction of logic programs by example-guided unfolding. J. Logic Program, 40: 159-183. DOI: 10.1016/S0743-1066(99)00017-5

Demba, M., F. Alexandre and K. Bsaies, 2005. Correction of faulty conjectures and programs extraction. Proceeding of the 20th International Workshop on Disproving Non-Theorems, Non-Validity, Non-Provability (CADE 05), July, 2005, pp: 17-27.

Ferri, C., J. Hernandez and M.J. Ramirez, 2001. Incremental learning of functional logic pro-grams. Lecturer Notes Comput. Sci., 2024: 233-247.

Flener, P. and Y. Deville, 1993. Logic program synthesis from incomplete specifications. J. Symb. Comput., 15: 775-805. DOI: 10.1016/S0747-7171(06)80012-X

Kanamori, T. and H. Seki, 1986. Verification of prolog programs using an extension of execution. Proceeding of the 3rd International Conference on Logic Programming, (LG'86), ACM Press, London, United Kingdom, pp: 475-489. http://portal.acm.org/citation.cfm?id=12105

Leuschel, M. and T. Massart, 2003. Inductive theorem proving by program specialization: Generating Proofs for Isabelle using Ecce (invited talk). Proceeding of the Symposium on Logic Based Program Synthesis and Transformation, Aug. 2003, Uppsala, Sweden, pp: 1-18. http://eprints.ecs.soton.ac.uk/8342/

Monroy, R., 2000. The use of abduction and recursion editor techniques for the correction of faulty conjectures. Proceeding of the 15th IEEE International Conference on Automated Software Engineering, Sept. 11-15, IEEE Computer Society Press, USA., pp: 91-99. http://portal.acm.org/citation.cfm?id=786768.786977

Protzen, M., 1996. Patching faulty conjectures. Lecturer Notes Comput. Sci., 1104: 77-91. DOI: 10.1007/3-540-61511-3_70

Sakurai, A. and H. Motoda, 1988. Proving definite clauses without explicit use of inductions. Lecturer Notes Comput. Sci., 383: 11-26. DOI: 10.1007/3-540-51564-X_52

Tamaki, H. and T. Sato, 1984. Unfold/fold transformation on logic programs. Proceeding of the 2nd International Conference on Logic Programming, July 1984, IEEE Computer Society Press, USA., pp: 127-138.