TR-176

Argus/v : A System for Verification of Prolog Programs

by
Tadashi Kanamori, Hiroshi Fujita
Kenji Horiuchi, Machi Maeji
(Mitsubishi Electric Corp.)
and
Hirohisa Seki (ICOT)

May, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Argus/V : A System for Verification of Prolog Programs

Tadashi KANAMORI[†], Hiroshi FUJITA[†]
Hirohisa SEKI[‡], Kenji HORIUCHI[†], Machi MAEJI[†]

[†] Mitsubishi Electric Corporation
Central Research Laboratory
Tsukaguchi-Honmachi 8-1-1
Amagasaki,Hyogo,JAPAN 661

[‡] ICOT Research Center
Institute for New Generation Computer Technology
Mita 1-4-28,Minato-ku
Tokyo,JAPAN 108

Abstract

The verification system Argus/V for proving properties of Prolog programs is outlined by contrasting verification with testing in logic programming. Specifications in Argus/V are given by a class of first order formulas, including goals for normal execution. Contrary to the specifications considered in the usual framework of verification, our specification states a partial property of the program than states what the progrom does as a whole. Though verification in Argus/V does not guarantee the correctness of the program at a stroke, the more properties of the program we prove, the closer the program is to what we intend. Verification in Argus/V is done using inference rules devised for verification, extension of these for usual execution. Execution in Prolog, on which testing is based, is a special case of the inferences in our verification. These two features of Argus/V show that both testing and verification are methods located on a continuous axis to confirm that the program is as we intend it.

Keywords : Verification, Testing, Theorem Proving, Prolog.

Contents

1. Introduction — Testing and Verification —

It is said that Prolog is a higher-level programming language than conventional languages, because operations in Prolog are further distanced from machine structure and closer to purely symbolic manipulation. Though such machine-independent human-oriented characteristics would seem to make programming easier, we still write incorrect Prolog programs quite frequently. For example, we might write the following incorrect program to reverse a list.

```
reverse([ ],[ ]).
reverse([X|L],M) :- reverse(L,N),append(N,X,M).
```

Figure 1. Incorrect Program for Reversing Lists

How can we determine whether the program is what we intend in our mind ? There are two well-known approaches, which give completely different impressions.

One approach is *testing*. Appropriate data is supplied to the program and it is run in order to see whether it shows unexpected responses or behavior. In general,*the more data for the program we test, the better we can determine how close the program is to what we intend it to be.* But testing is considered a rather naive and informal method, and troublesome to the human programmer.

Another approach is *verification*. A specification, that shows *what* the program does as a whole, is given and proved logically with respect to the program, that shows *how* the desired resuls are computed. Verification is considered a rather formal method, to confirm the correctness of the program at a stroke, possibly mechanically.

These approaches seem to be fundamentally separated in conventional programming. How do things stand in logic programming ? In order to answer the question and make the motivation of our verification method understandable, we review what we are doing in testing in Section 2 and the paradigm of logic programming in Section 3. Then an outline of our verification system Argus/V is given in Section 4. In particular, we emphasize the characteristics of specifications and the use of extended execution. In Section 5, we discuss the similarities between testing and our verification with respect to two features, that is, both approaches confirm the correctness of programs gradually and both approaches are based on execution.

2. Testing of Prolog Programs

Let us test the *reverse* program in Figure 1. *Tracer*, one of the testing tools of DEC10 Prolog system [23], responds as follows.

```
| ?- reverse([2,1],M).
    (1) 0 Call : reverse([2,1],_40) ?
    (2) 1 Call : reverse([1],_105) ?
    (3) 2 Call : reverse([ ],_117) ?
    (3) 2 Exit : reverse([ ],[ ])
    (4) 2 Call : append([ ],1,_105) ?
    (4) 2 Call : append([ ],1,1)
    ⋮
```

Figure 2.1. Example of A Testing of Prolog Program

A goal in the trace holds, when the goals (with "Call") below the goal hold. By observing the behavior in the trace and finding the goal contradicting the programmer's intention, the programmer can correct the program in Figure 1 as follows.

```
reverse([ ],[ ]).
reverse([X|L],M) :- reverse(L,N),append(N,[X],M).
```

Figure 2.2. Correct Program for Reversing Lists

After all, *testing is confirming whether there is any difference between the model the programmer has in his/her mind and the actual behavior of the program.*

3. Logic Programming Paradigm Revisited

Now, let us recall the paradigm of logic programming, the central idea of the Japanese Fifth Generation Computer Systems project. *The execution of a Prolog program is the construction of a logical proof* ([1],[9],[10],[15],[20]). For example, the execution of ?-$reverse([2,1],M)$ is the construction of a proof of $\exists M reverse([2,1],M)$. In general, we have the following inference rule for each definite clause "B :- $B_1, B_2, \ldots, B_m$", where σ is an m.g.u. of A and B. (It is read the formula below the line holds when the formulas above the line hold.)

$$\text{execution} \qquad \frac{\sigma(B_1 \wedge B_2 \wedge \cdots \wedge B_m)}{A}$$

Figure 3.1. Inference Rule for Execution

The proof of a given formula we would like to prove is constructed from the bottom. (See the Figure 3.2 below.) The tracer in Section 2 shows the proof upside-down from left to right.

$$\cfrac{\cfrac{\text{reverse([ ],[ ])}\wedge\text{append([ ],1,1)}}{\text{reverse([1],1)}\wedge\text{append(1,2,?)}}}{\text{reverse([2,1],?)}}$$

Figure 3.2. Proof Tree Corresponding to Execution

According to the paradigm, the behavior on which we focus our attention in testing is how the proof of formulas of the form $\exists Y_1, Y_2, \ldots, Y_m\ (A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ is constructed.

4. Verification of Prolog Programs

4.1. Specification of Prolog Programs

The most prominent feature of logic programming languages is, of course, the close relation to logic, the origin of the name. For example, Prolog is very close to, actually a sublogic of, first order logic, long used as one of the most common specification languages. This close relation between programming languages and specification languages prompts reexamination of the nature and the role of specifications and verification.

Specification and verification are still necesary.

Some people think that Prolog programs are formulas of first order logic, rendering independent specification and verification redundant. We agree that in some cases a specification can be a Prolog program as it is, and we cannot write any other simpler specifications of some programs. But, as can be easily seen, Prolog programs are not always specifications. Many computation mechanism have been devised to increase efficiency. Moreover, even if a Prolog program is a description of our intention in logical formulas, it is written from one point of view, which might be erroneous. Confirmation that the program is what we intend must be effected from another point of view by testing or verification [7].

Specifications verified might be partial.

It has often been said that specifications are sometimes as large as programs themselves. The specifications considered so far are usually *total*, that is, they must contain all the information about what the program does as a whole. Such specifications are necessary for program synthesis, because specifications for synthesis usually have to contain all the information for the program to be constructed. The close relation between Prolog and first order logic suggests the possibility of relaxing this restriction on verification. The specifications in our verification system might be *partial*, that is, they are not necessarily the total description of what the program does as a whole, because the program itself may be a part of its own specification. For example, the following property *reverse-reverse*

∀ X,Y (reverse(X,Y) ⊃reverse(Y,X))

is also satisfied by the identity relation *id* defined by

id(X,X).

Hence, even if we have proved the *reverse-reverse* property with respect to the program at hand, we can't conclude that our program is the correct *reverse*. It is a fortunate situation when we are able to write down the total specification. But we usually content ourselves with a set of partial specifications. *The more properties of the program we verify, the closer the program is to what we intend.* After all, *verification is confirming whether there is any difference between the model the programmer has in his/her mind and the actual properties of the program.* Our verification is much closer to testing in its nature and functioning.

Specification Formulas and Goal Formulas

Now we introduce the class of first order formulas used for specification in Argus/V.

We generalize the distinctions of positive and negative goals. The *positive* and *negative* *subformulas* of a formula $\mathcal{F}$ are defined as follows (see [24],[21],[22],[25]).

(a) $\mathcal{F}$ is a positive subformula of $\mathcal{F}$.

(b) When $\neg\mathcal{G}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ is a negative (positive) subformula of $\mathcal{F}$.

(c) When $\mathcal{G}\wedge\mathcal{H}$ or $\mathcal{G}\vee\mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ and $\mathcal{H}$ are positive (negative) subformulas of $\mathcal{F}$.

(d) When $\mathcal{G}\supset\mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}$ is a negative (positive) subformula of $\mathcal{F}$ and $\mathcal{H}$ is a positive (negative) subformula of $\mathcal{F}$.

(e) When $\forall X\mathcal{G}$ or $\exists X\mathcal{G}$ is a positive (negative) subformula of $\mathcal{F}$, then $\mathcal{G}_X(t)$ is a positive (negative) subformula of $\mathcal{F}$.

Example 4.1.1. Let $\mathcal{F}$ be

$\forall$ B,U,A_1,V,A (reverse(B,[U|A_1])$\wedge$append(A_1,[V],A) $\supset\exists$ A_2 (reverse(B,A_2)$\wedge$append(A_2,[V],[U|A])))

Then $\exists A_2(\mathit{reverse}(B,A_2)\wedge \mathit{append}(A_2,[V],[U|A]))$ is a positive subformula of $\mathcal{F}$, while $\mathit{reverse}(B,[U|A_1])$ is a negative subformula of $\mathcal{F}$.

Let $\mathcal{F}$ be a closed first order formula. When $\forall X\mathcal{G}$ is a positive subformula or $\exists X\mathcal{G}$ is a negative subformula of $\mathcal{F}$, X is called a *free variable* of $\mathcal{F}$. When $\forall Y\mathcal{H}$ is a negative subformula or $\exists Y\mathcal{H}$ is a positive subformula of $\mathcal{F}$, Y is called an *undecided variable* of $\mathcal{F}$. In other words, free variables are variables quantified universally, and undecided variables are those quantified existentially when $\mathcal{F}$ is converted to its prenex normal form.

Example 4.1.2. Let $\mathcal{F}$ be

$\forall$ B,U,A_1,V,A (reverse(B,[U|A_1])$\wedge$append(A_1,[V],A) $\supset\exists$ A_2 (reverse(B,A_2)$\wedge$append(A_2,[V],[U|A])))

Then B,U,A_1,V and A are all free variables, while A_2 is an undecided variable.

A closed first order formula S is called a *specification formula* (or *S-formula* for short) when

(a) no free variable in S is quantified in the scope of quantification of an undecided variable in S and

(b) each undecided variable appears only in some positive conjunction of atoms $A_1\wedge A_2\wedge\cdots\wedge A_k$ in S.

In other words, S-formulas are formulas convertible to prenex normal form $\forall X_1,X_2,\ldots,X_n$ $\exists Y_1,Y_2,\ldots,Y_m\mathcal{F}$ and none of $Y_1,Y_2,\ldots,Y_m$ appears only in some positive conjunction of atoms in $\mathcal{F}$. Note that S-formulas include both universal formulas $\forall X_1,X_2,\ldots,X_n\mathcal{F}$ and usual execution goals $\exists Y_1,Y_2,\ldots,Y_m$ $(A_1\wedge A_2\wedge\cdots\wedge A_k)$.

Example 4.1.3. Let S be

$\forall$ B,U,A_1,V,A (reverse(B,[U|A_1])$\wedge$append(A_1,[V],A) $\supset\exists$ A_2 (reverse(B,A_2)$\wedge$append(A_2,[V],[U|A])))

Then S is an S-formula, because free variables B,U,A_1,V and A are quantified outside $\exists A_2$, and A_2 appears only in the positive conjunction $\mathit{reverse}(B,A_2)\wedge\mathit{append}(A_2,[V],[U|A])$. An execution goal

$\exists$ C append([1,2],[3],C)

is also an S-formula.

A formula G obtained from an S-formula S by leaving free variable X as it is, replacing undecided variable Y with $?Y$ and deleting all quantifications is called a *goal formula* of S. Note that S can be uniquely restorable from G. In the following, we use goal formulas instead of original S-formulas. Goal formulas are denoted by F,G,H.

Example 4.1.4. An S-formula

$\forall$ B,U,A_1,V,A (reverse(B,[U|A_1])$\wedge$append(A_1,[V],A) $\supset\exists$ A_2 (reverse(B,A_2)$\wedge$append(A_2,[V],[U|A])))

is represented by a goal formula

reverse(B,[U|A_1])$\wedge$append(A_1,[V],A) $\supset$reverse(B,?A_2)$\wedge$append(?A_2,[V],[U|A]).

An execution goal

$\exists$ C append([1,2],[3],C)

is represented by a goal formula

append([1,2],[3],?C).

Let S be a specification in an S-formula, M_0 be the minimum Herbrand model [10] of P and P^* be the completion [8] of P. We adopt a formulation as follows : Model-theoretically speaking, verification of S with respect to P is showing $M_0 \models S$. Proof-theoretically speaking, it is proving S from P^* using first order inference and some induction. (Of course, the proof-theoretical formulation is weaker than the model-theoretical formulation. See Section 4.4 for induction.)

4.2. Inference Rules for Verification

Though we have pointed out the similarity between testing and our verification, it is meaningless to just rephrase the definition of verification. The current Prolog interpreter can't execute our S-formulas directly. Now we present the method and the mechanism of our verification (cf. [3],[13],[14],[28]). Our inference rules for verification consist of *extended execution* and *computational induction*. Extended excution is an extension of usual execution and consists of case splittings ($\wedge$-deletion,$\vee$-deletion and $\supset$-deletion), definite clause inference (DCI), "Negation as Failure" inference (NFI) and simplification. We omit discussion of the case splitting rules in what follows, because they are not used very frequently. See [Kanamori and Seki 1985], [Kanamori 1986] for details.

Using intuitive notations, DCI, NFI and simplification are depicted as inference rules as follows. In the followings, we use $\mathcal{F}_{\mathcal{G}}(\mathcal{H})$ for replacement of *all* occurrences of a formula $\mathcal{G}$ in a formula $\mathcal{F}$ with $\mathcal{H}$ and $\mathcal{F}_{\mathcal{G}}[\mathcal{H}]$ for replacement of an occurrence of a formula $\mathcal{G}$ in a formula $\mathcal{F}$ with $\mathcal{H}$. See the following explanation for meanings of other notations.

DCI $$\frac{\sigma(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])}{G_+[A]}$$

NFI $$\frac{\tau_1(G_A[\wedge_{i=1}^{m_1} B_{1i}]) \quad \cdots \quad \tau_k(G_A[\wedge_{i=1}^{m_k} B_{ki}]) \quad G_A[false]}{G_-[A]}$$

simplification $$\frac{\sigma(G)_A(true) \quad \sigma(G)_A(false)}{G}$$

Figure 4.2. Main Inference Rules for Verification

4.3. Extended Execution

The execution of positive goals is generalized using polarity.

Definite Clause Inference(DCI)

Let A be a positive atom in a goal formula G and "B :- $B_1, B_2, \ldots, B_m$" be any definite clause in P. When A is unifiable with B by an m.g.u. σ without instantiation of free variables, a new OR-goal $\sigma(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])$ is generated. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as fresh undecided variables.

Example 4.3.1. Let S be

$\forall$ A,B,C,U ((reverse(C,B)$\supset$reverse(B,C))$\supset$(reverse(A,[U|B])$\supset$reverse([U|B],A))).

Then the goal formula of S is

(reverse(C,B) $\supset$reverse(B,C)) $\supset$(reverse(A,[U|B]) $\supset$reverse([U|B],A))

We can apply DCI to *reverse*$([U|B], A)$ and it is replaced with *reverse*$(A, ?D) \wedge$ *append*$(?D, [U], B)$. Note that the variable in the body is treated as an undecided variable $?D$.

Example 4.3.2. When S is an existential formula of the form $\exists Y_1 Y_2 \cdots Y_m (A_1 \wedge A_2 \wedge \cdots \wedge A_k)$, i.e., of the form of usual execution goals, the goal formula of S is $?\text{-}A_1, A_2, \ldots, A_k$. (The juxtaposition delimited by "," denotes conjunction and $?\text{-}G$ denotes the goal formula obtained by replacing every variable Y in G with $?Y$.) Then usual execution is applied to $?\text{-}$ $A_1, A_2, \ldots, A_k$. The figure below shows an example, where *common* and *reverse* are defined by

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

```
          common(1,[1,2],[3,1])
                    |
   member(1,[1,2]),member(1,[3,1])
                    |
            member(1,[3,1])
                    |
             member(1,[1])
                    |
                  true
```

Figure 4.3.1. Definite Clause Inference for Usual Positive Goals

We also generalize the execution of negative goals using polarity.

"Negation as Failure" Inference(NFI)

Let A be a negative atom in a goal formula G. We generate new AND-goals $\tau(G_A[B_1 \wedge B_2 \wedge \cdots \wedge B_m])$ for every definite clause "B :- $B_1, B_2, \ldots, B_m$" in P, whose head B is unifiable with A, and an AND-goal $G_A[\mathit{false}]$. ($B_1 \wedge B_2 \wedge \cdots \wedge B_m$ is *true* when $m = 0$.) All new variables introduced are treated as fresh free variables. (Note that A always includes only free variables and τ may be any m.g.u. without restriction.)

Example 4.3.3. Let S be

$\forall$ A,B,C,U ((reverse(A,C)$\supset$reverse(C,A))$\supset$(reverse([U|A],B)$\supset$reverse(B,[U|A]))).

Then the goal formula of S is

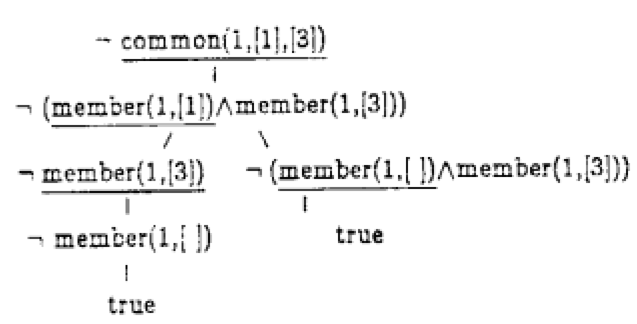
(reverse(A,C) $\supset$reverse(C,A)) $\supset$(reverse([U|A],B) $\supset$reverse(B,[U|A]))

We can apply NFI to *reverse*$([U|A], B)$. In the first goal,the atom is replaced with *reverse*$(A, D) \wedge$ *append*$(D, [U], B)$. Note that the variable in the body is treated as a free variable D. The last goal obtained by replacing the atom with *false* is trivially *true*.

Example 4.3.4. Let S be a specification of the form $\neg A$ where A is a ground atom. Suppose there exist k definite clauses whose heads are unifiable with A by m.g.u.s $\tau_1, \tau_2, \ldots, \tau_k$. When NFI is applied to A, we have $k+1$ AND-goals

$\neg\tau_1(B_{11}\wedge B_{12}\wedge\cdots\wedge B_{1m_1})$,
$\neg\tau_2(B_{21}\wedge B_{22}\wedge\cdots\wedge B_{2m_2})$,
$\vdots$,
$\neg\tau_k(B_{k1}\wedge B_{k2}\wedge\cdots\wedge B_{km_k})$,
$\neg false$.

The last goal formula is trivially *true*. Other goal formulas are of the form $\forall X_1, X_2, \ldots, X_n$ $\neg(A_1\wedge A_2\wedge\cdots\wedge A_m)$, because variables introduced from the bodies of the definite clauses are free variables in the generated goal formulas. We can continue applying NFI by selecting atoms in each body of the goal formula. When a selected atom has no unifiable head, the only goal formula generated is the last one, which is always *true*. When all goal formulas are reduced to *true*, $\neg A$ is proved. This is exactly the "Negation as Failure" rule in the usual sense [8],[15]. The figure below shows an example.



¬ common(1,[1],[3])
¬ (member(1,[1])∧member(1,[3]))
¬ member(1,[3])
¬ (member(1,[ ])∧member(1,[3]))
¬ member(1,[ ])
true
true

Figure 4.3.2. "Negation as Failure" Inference for Usual Negative Goals

We sometimes simplify goal formulas by assuming that some atom is *true* or *false* (cf.[22]).

Simplification

Let G be a goal formula. When $A_1, A_2, \ldots, A_m$ are positive atoms and $A_{m+1}, A_{m+2}, \ldots, A_n$ are negative atoms unifiable to A by an m.g.u. σ without instantiation of free variables $(0 < m < n)$, we generate new AND-goals $\sigma(G)_A(true)$ and $\sigma(G)_A(false)$.

In the following examples, both σ are $<>$ and undecided variables are not instantiated. For more general simplifications with instantiation of undecided variables, see Figure 4.3.3.

Example 4.3.5. Let G be a goal formula

(add(X,Y,Z) ⊃add(Y,X,Z)) ⊃(add(X,Y,Z) ⊃add(Y,s(X),s(Z)))

of an S-formula

∀ X,Y,Z ((add(X,Y,Z) ⊃add(Y,X,Z)) ⊃(add(X,Y,Z) ⊃add(Y,s(X),s(Z)))).

Because $\sigma = <>$ is a substitution without instantiation of free variables and unifies the positive atom $add(X,Y,Z)$ and the negative atom $add(X,Y,Z)$, we generate new AND-goals

(true ⊃add(Y,X,Z)) ⊃(true ⊃add(Y,s(X),s(Z))) ,
(false ⊃add(Y,X,Z)) ⊃(false ⊃add(Y,s(X),s(Z))) ,

i.e., $add(Y,X,Z)\supset add(Y,s(X),s(Z))$ and *true*. This inference corresponds to generating

(Y+X)+1=Y+(X+1)

from

X+Y=Y+X ⊃(X+Y)+1=Y+(X+1)

in functional programs, i.e., using the equation $X+Y=Y+X$ in the premise and throwing it away. This is called *cross-fertilisation* in the Boyer Moore Theorem Prover (BMTP) [5].

Example 4.3.6. Let G be a goal formula

$$(\mathrm{reverse}(A,C) \supset \mathrm{reverse}(C,A)) \supset (\mathrm{reverse}(A,C) \wedge \mathrm{append}(C,[U],B) \supset \mathrm{reverse}(B,[U|A]))$$

of an S-formula

$$\forall A,B,C,U\ ((\mathrm{reverse}(A,C) \supset \mathrm{reverse}(C,A)) \supset ((\mathrm{reverse}(A,C) \wedge \mathrm{append}(C,[U],B)) \supset \mathrm{reverse}(B,[U|A]))).$$

Because $\sigma = <>$ is a substitution without instantiation of free variables and unifies the positive atom $\mathit{reverse}(A,C)$ and the negative atom $\mathit{reverse}(A,C)$, we generate new AND-goals

$$(\mathrm{true} \supset \mathrm{reverse}(C,A)) \supset (\mathrm{true} \wedge \mathrm{append}(C,[U],B) \supset \mathrm{reverse}(B,[U|A]))\ ,$$
$$(\mathrm{false} \supset \mathrm{reverse}(C,A)) \supset (\mathrm{false} \wedge \mathrm{append}(C,[U],B) \supset \mathrm{reverse}(B,[U|A]))\ ,$$

i.e., $\mathit{reverse}(C,A) \supset (\mathit{append}(C,[U],B) \supset \mathit{reverse}(B,[U|A]))$ and **true**. This inference corresponds to generating

$$\mathrm{reverse}(C)=A \supset \mathrm{reverse}(\mathrm{append}(C,[U]))=[U|A]$$

from

$$\mathrm{reverse}(\mathrm{reverse}(A))=A \supset \mathrm{reverse}(\mathrm{append}(\mathrm{reverse}(A),[U]))=[U|A]$$

in functional programs, i.e., replacement of the special term $\mathit{reverse}(A)$ with a variable C. This is called *generalization* in BMTP [5].

The Figure 4.3.3. below is one of the sequences of the applications of extended execution. A goal in the sequence holds, when the goals below it hold, like the goals in Figure 2.1.

$$\begin{array}{l}
\mathrm{reverse}(M,[X|L_1]) \wedge \mathrm{append}(L_1,[Y],L) \supset \underline{\mathrm{reverse}([Y|M],[X|L])} \\
\quad \Downarrow \text{DCI with } <> \\
\mathrm{reverse}(M,[X|L_1]) \wedge \mathrm{append}(L_1,[Y],L) \supset \mathrm{reverse}(M,?L_2) \wedge \underline{\mathrm{append}(?L_2,[Y],[X|L])} \\
\quad \Downarrow \text{DCI with } <?L_2 \Leftarrow [X|?L_3]> \\
\underline{\mathrm{reverse}(M,[X|L_1])} \wedge \mathrm{append}(L_1,[Y],L) \supset \underline{\mathrm{reverse}(M,[X|?L_3])} \wedge \mathrm{append}(?L_3,[Y],L) \\
\quad \Downarrow \text{simplification with } <?L_3 \Leftarrow L_1> \\
\underline{\mathrm{append}(L_1,[Y],L)} \supset \underline{\mathrm{append}(L_1,[Y],L)} \\
\quad \Downarrow \text{simplification with } <> \\
\mathrm{true}
\end{array}$$

Figure 4.3.3. Example of Verification of Prolog Programs

4.4. Computational Induction

Because we have adopted the model-theoretical formulation in 4.1, we need to use some kind of induction in order to make our proof system as strong as possible to approximate the model-theoretic formulation.

The following induction scheme is used for induction on natural numbers [6].

$$\frac{Q(0) \qquad \forall X\ (Q(X) \supset Q(X+1))}{\forall X\ (\mathrm{number}(X) \supset Q(X))}$$

Figure 4.4.1. Induction Scheme for Natural Number

Note that the *number* predicate is defined in Prolog as in Figure 4.4.2 and the induction formulas above the line in Figure 4.4.1 is exactly what are obtained by replacing *number* in Figure 4.4.2 with Q.

```
number(0).
number(X+1) :- number(X).
```

Figure 4.4.2. Prolog Programs Defining Natural Number

Similarly, the induction scheme for *reverse* is obtained by replacing *reverse* in Figure 2.2. with Q. This is de Bakker and Scott's computational induction for Prolog ([2],[9],[11],[12],[29]).

$$\frac{Q([\,],[\,]) \qquad \forall L,M,N,X\ (Q(L,N) \wedge \mathrm{append}(N,[X],M) \supset Q([X|L],M))}{\forall L,M\ (\mathrm{reverse}(L,M) \supset Q(L,M))}$$

Figure 4.4.3. Induction Scheme for *reverse*

Example 4.4. When the induction scheme above is applied to *reverse-reverse*, i.e.,

reverse(L,M) ⊃reverse(M,L)

the following two induction formulas are generated.

reverse([ ],[ ])

reverse(N,L)∧append(N,[X],M) ⊃reverse(M,[X|L]).

In general, the goals we are going to prove are not necessarily of the form $p(X_1, X_2, \ldots, X_n) \supset Q(X_1, X_2, \ldots, X_n)$. Moreover, more than two induction schemes might be suggested. In order to manage such situations, we have a device to generate and manipulate induction schemes based on an equivalence-preserving program transformation [27]. See [Kanamori and Fujita 1986] for details.

4.5. Argus/V Verification System

The Argus/V verification system has the module structure depicted below. A given specification is first converted to its goal formula, then passed through the four modules in Figure 4.5.

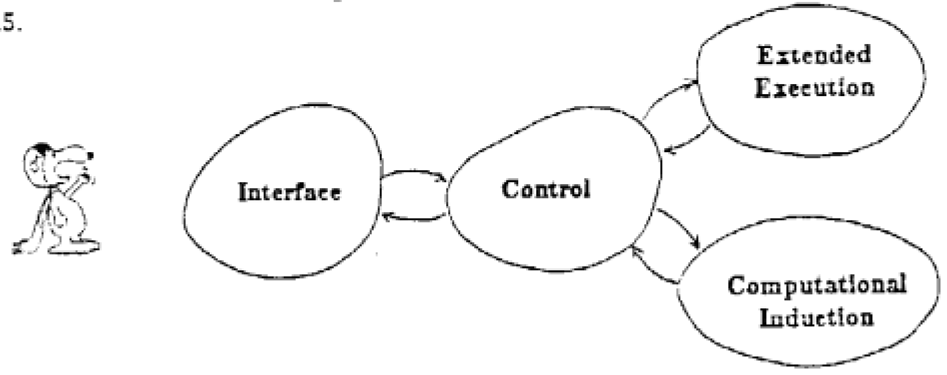


Figure 4.5. Argus/V Module Structure

Because we have additional and generalized inference rules for verification, we need a procedure to select the inference rules to be applied. When and how extended execution and computational induction are applied is controlled by many BMTP-like heuristics [5]. This can be considered a kind of meta-inference [4],[26]. See [Kanamori and Horiuchi 1985] for the use of type inference in Argus/V.

5. Discussion — Testing and Verification —

As shown in Section 4, verification in Argus/V is very close to testing not only in its nature and the role it plays but also in the methodology and its mechanism. In fact, testing is a special case of verification in Argus/V in which the specifications are ground goals or existentially quantified atoms. The difference is that verification in Argus/V is more general than testing. For example, testing can confirm *reverse-reverse* for only lists with specific lengths, while verification in Argus/V proves it for lists of any length. In a sense, testing deals with the superficial observable *directly*, while verification penetrate the interior observable only *indirectly*. In other words,"our *prover* is a *prober*."

6. Conclusions

We have given an outline of the Argus/V verification system for proving properties of Prolog programs by contrasting verification with testing in logic programming. The first version of Argus/V was developed between April 1984 and March 1985. It consists of about 7000 lines in DEC-10 Prolog and takes about 9.5 seconds (CPU time of DEC2060 with 384 kw main memory) to prove *reverse-reverse* automatically. More than 50 theorems have already been proved automatically and the number is increasing.

Acknowledgements

The Argus/V verification system is a subproject of the Fifth Generation Computer System(FGCS) "Intelligent Programming System". The authors would like to thank Dr.K.Fuchi (Director of ICOT) for the opportunity of doing this research and Dr.K.Furukawa (Chief of ICOT 1st Laboratory) and Dr.T.Yokoi (Chief of ICOT 2nd Laboratory) for their advice and encouragement.

References

[1] Apt,K.R. and M.H.van Emden, "Contribution to the Theory of Logic Programming", J.ACM, Vol.29, No.3, pp.841-862, 1982.

[2] de Bakker,J.W. and D.Scott, "A Theory of Programs", Unpublished Notes, IBM Seminar, Vienna, 1969.

[3] Bowen,K.A., "Programming with Full First-Order Logic", Machine Intelligence 10 (J.E.Hayes, D.Michie and Y-H.Pao Eds), pp.421-440, 1982.

[4] Bowen,K.A. and R.A.Kowalski, "Amalgamating Language and Metalanguage in Logic Programming", in Logic Programming (K.L.Clark and S-Å.Tärnlund Eds), Academic Press, 1980.

[5] Boyer,R.S. and J.S.Moore, "Computational Logic", Academic Press, 1979.

[6] Burstall,R., "Proving Properties of Programs by Structural Induction", Comput.J., Vol.12, No.1., pp.41-48, 1969.

[7] Clark,K.L. and S-Å.Tärnlund, "A First Order Theory of Data and Programs", in Information Processing 77 (B.Gilchrist Ed), pp.939-944, 1977.

[8] Clark,K.L., "Negation as Failure", in Logic and Database (H.Gallaire and J.Minker Eds),pp.293-302,1978.

[9] Clark,K.L., "Predicate Logic as a Computational Formalism", Chap.4, Research Monograph : 79/59, TOC, Imperial College, 1979.

[10] van Emden,M.H. and R.A.Kowalski, "The Semantics of Predicate Logic as a Programing Language", J.ACM, Vol.23, No.4, pp.733-742, 1976.

[11] Gordon,M.J.,A.J.Milner and C.P.Wadsworth, "Edinburgh LCF — A Mechanized Logic of Computation", Lecture Notes in Computer Science 78, Springer, 1979.

[12] Hagiya,M. and T.Sakurai, "Foundation of Logic Programming Based on Inductive Definition", New Generation Computing, Vol.2, pp.59-77, 1984.

[13] Hansson,A. and S-Å.Tärnlund, "A Natural Programming Calculus", Proc.of 6th International Joint Conference on Artificial Intelligence, pp.348-355,1979.

[14] Haridi,S. and D.Sahlin, "Evaluation of Logic Programs Based on Natural Deduction", Proc.of 2nd Workshop on Logic Programming, 1983.

[15] Jaffar,J.,J-L.Lassez and J.Lloyd, "Completeness of the Negation as Failure Rule", Proc.of 8th International Joint Conference on Artificial Intelligence, Vol.1, pp.500-506, 1983.

[16] Kanamori,T.and H.Seki, "Verification of Prolog Programs Using An Extension of Execution", ICOT Technical Report, TR-096, 1984. Also Proc.of of 3rd International Conference on Logic Programming, 1986.

[17] Kanamori,T.and H.Fujita, "Formulation of Induction Formulas in Verification of Prolog Programs", ICOT Technical Report, TR-094, 1984. Also Proc.of Conference on Automated Deduction, 1986.

[18] Kanamori,T.and K.Horiuchi, "Type Inference in Prolog and Its Applications", ICOT Technical Report, TR-095, 1984. Also Proc.of 9th International Joint Conference on Artificial Intelligence, pp.704-707, Los Angeles, 1985.

[19] Kanamori,T., "Soundness and Completeness of Extended Execution for Proving Properties of Prolog Programs", ICOT Technical Report, to appear, 1986.

[20] Kowalski,R.A., "Logic for Problem Solving", Chap.10-12, North Holland, 1980.

[21] Manna,Z.and R.Waldinger, "A Deductive Approach to Program Synthesis", ACM Trans. on Programming Languages and Systems, Vol.2, No.1, pp.90-121, 1980.

[22] Murray,N.V., "Completely Non-Clausal Theorem Proving", Artificial Intelligence, Vol.18, pp.67-85, 1982.

[23] Pereira,L.M.,F.C.N.Pereira and D.H.D.Warren, "User's Guide to DECsystem-10 Prolog", Occasional Paper 15, Dept.of Artificial Intelligence, Edinburgh,1979.

[24] Prawitz,D., "Natural Deduction,A Proof Theoretical Study", Almqvist & Wiksell, Stockholm, 1965.

[25] Schütte,K., "Proof Theory", (translated by J.N.Crossley), Springer Verlag, 1977.

[26] Stering,L. and A.Bundy, "Meta-Level Inference and Program Verification", in 6th Automated Deduction (W.Bibel Ed), Lecture Notes in Computer Science 138, pp.144-150, 1982.

[27] Tamaki,H. and T.Sato, "Unfold/Fold Transformation of Logic Programs", Proc.of 2nd International Logic Programming Conference, pp.127-138, 1984.

[28] Tärnlund,S-Å., "Logic Programming Language Based on A Natural Deduction System", UPMAIL Technical Report, No.6, 1981.

[29] Weyrauch,R.W. and R.Milner, "Program Correctness in A Mechanized Logic", Proc.of 1st USA-Japan Computer Conference, 1972.