Logic Program Testing Based on Abstract Interpretation*

Lunjin Lu and Peter Greenfield

The University of Birmingham, Birmingham B15 2TT, U.K.

Abstract. In this paper we present a logic program testing algorithm. This algorithm tests a program against a particular program property each time it is applied. A program property is derived from the program and is compared with the desired program property and the difference between them is used to guide test data generation.

1 Introduction

A logic program is intended to compute a set of ground atoms each of which falls into a relation. When a logic program has been written, it is necessary to make sure that the program computes the right set of ground atoms. The set of ground atoms that the program is intended to compute is called the intended semantics of the program whilst the set of ground atoms that the program actually computes is called the actual semantics of the program. When the actual semantics of the program is not equal to its intended semantics, the program contains at least one bug symptom. A ground atom in the actual semantics but not in the intended semantics is called an inconsistency symptom and a ground atom in the intended semantics but not in the actual semantics is called an insufficiency symptom.

While program testing has been an intensive research topic in the imperative programming community, it does not appear to have received much attention in the logic programming community; we have found little literature on logic program testing. We propose a logic program testing algorithm that tests the program against program properties. Each time the program is tested, a program property such as the type of the program is derived from the program and this derived program property is then compared with the desired program property. The difference between the derived program property and the desired program property is then used to limit the range from which a set of test data is selected. The motivation is twofold. Firstly, testing the program with all possible input values is infeasible because there is usually infinite number of input values. Secondly, the bugs in the program have manifestations in the properties of the program. We derive program properties from the program by means of abstract interpretation and hence we refer to program properties as abstract semantics. The derived program property will be called the actual abstract semantics of the program and the desired program property will be called the intended abstract semantics of the program.

* Appeared in Lecture Notes in Computer Science, 735:170-180, Springe-Verlag, 1993

2 Semantics and Abstract Interpretation

A language L is the set of programs that can be constructed from the grammar rules of L. The semantics of a program in L is a point in a semantic domain $\mathcal{D}$ and is given by a semantic function $\mu : L \mapsto \mathcal{D}$. Taking the definite logic program language as an example, L is the set of all definite logic programs, the semantic domain $\mathcal{D}$ is the powerset of the Herbrand base of L and the semantic function μ is the least fixpoint semantic function $\lambda P.lfp \circ T_P$ given by van Emden and Kowalski [21] where lfp is the least fixpoint operator.

A semantic domain $\mathcal{D}$ is a complete lattice $\mathcal{D}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $\sqsubseteq$ is a partial order on $\mathcal{D}$ such that any subset X of $\mathcal{D}$ has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$ and $\bot = \sqcup\emptyset = \sqcap\mathcal{D}$ and $\top = \sqcap\emptyset = \sqcup\mathcal{D}$ [19]. A semantic domain $\mathcal{D}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is usually a powerset domain. $\mathcal{D} = \wp(D)$ for some base domain D, say the Herbrand base of the definite logic program language, $\sqsubseteq$ is set inclusion $\subseteq$, $\bot$ is the empty set $\emptyset$, $\top$ is the base domain D, $\sqcup$ is set union $\cup$ and $\sqcap$ is set intersection $\cap$.

The semantic domains that we consider in this paper are powerset domains. We now define the notion of the correctness of a program with respect to a semantic function and the intended semantics of the program on the corresponding semantic domain.

Definition 1. Let P be a program in P, $\mu : L \mapsto \mathcal{D}$ a semantic function and I the intended semantics of P on $\mathcal{D}$. When $\mu(P) = I$, P is said to be correct wrt μ and I. When $\mu(P) \neq I$, P is incorrect wrt μ and I. P is said to be inconsistent wrt μ and I if $\mu(P) - I \neq \emptyset$. Any $A \in \mu(P) - I$ is said to be an inconsistency symptom of P wrt μ and I. P is said to be insufficient wrt μ and I if $I - \mu(P) \neq \emptyset$. Any $A \in I - \mu(P)$ is said to be an insufficiency symptom of P wrt μ and I.

Abstract interpretation is a general scheme for static program analysis. The basic idea is that program analysis consists in approximating the standard semantic function $\mu_c : L \mapsto \mathcal{D}_c$ by a non-standard semantic function $\mu_a : L \mapsto \mathcal{D}_a$. The standard domain $\mathcal{D}_c$ is called the concrete domain, and the standard semantic function $\mu_c : L \mapsto \mathcal{D}_c$ is called the concrete semantic function. The non-standard domain $\mathcal{D}_a$ is called the abstract domain and the non-standard semantic function $\mu_a : L \mapsto \mathcal{D}_a$ is called the abstract semantic function. The foundation of abstract interpretation has been laid by Cousot and Cousot [5, 6]. In the logic programming community, Mellish [16], Jones et al. [10], Bruynooghe [4] and Marriott et al. [14] have developed several frameworks for abstract interpretation of logic programs and found many applications of these frameworks such as in occur check [20], mode inference [15, 16, 3] and type inference [4, 3].

Let the concrete semantic function be $\mu_c : L \mapsto \mathcal{D}_c(\sqsubseteq_c, \bot_c, \top_c, \sqcup_c, \sqcap_c)$ and the abstract semantic function be $\mu_a : L \mapsto \mathcal{D}_a(\sqsubseteq_a, \bot_a, \top_a, \sqcup_a, \sqcap_a)$. If there is an abstraction function $\alpha : \mathcal{D}_c \mapsto \mathcal{D}_a$ and a concretization function $\gamma : \mathcal{D}_a \mapsto \mathcal{D}_c$ such that $\alpha \circ \mu_c \sqsubseteq_a \mu_a$ and $\mu_c \sqsubseteq_c \gamma \circ \mu_a$, then μ_a is called an abstract interpretation of L relative to μ_c and (α, γ). We assume that $\mathcal{D}_c(\sqsubseteq_c, \bot_c, \top_c, \sqcup_c, \sqcup_c) = \wp(D_c)(\subseteq, \emptyset, D_c, \cup, \cap)$ and $\mathcal{D}_a(\sqsubseteq_a, \bot_a, \top_a, \sqcup_a, \sqcup_a) = \wp(D_a)(\subseteq, \emptyset, D_a, \cup, \cap)$ since both $\mathcal{D}_c$ and $\mathcal{D}_a$ are powerset domains. We also assume that α is a setwise function from D_c to D_a and γ is a setwise function from D_a to $\wp(D_c)$ such that $\gamma(x) = \{y | y \in D_c \textit{ and } \alpha(y) = x\}$. Definition 1 applies to both the concrete semantics by letting $\mu = \mu_c$ and $I = I_c$ and the abstract semantics by letting $\mu = \mu_a$ and $I = I_a$.

Each of $\mu_c(P)$ and I_c is usually a set of infinite size. This renders it infeasible to compare $\mu_c(P)$ with I_c. It is possible to compare $\alpha \circ \mu_c(P)$ with $\alpha \circ I_c$ if D_a is finite.

3 Algorithm

In this section, we discuss the relationship between bug symptoms of P wrt μ_c and I_c and bug symptoms of P wrt μ_a and I_a and present a testing algorithm for logic programs.

Lemma 2. *Let μ_a be an abstract interpretation relative to μ_c and (α, γ).*

(1) *If $\alpha \circ \mu_c(P) - \alpha \circ I_c \neq \emptyset$, then P is inconsistent* wrt *μ_c and I_c. For each inconsistency symptom $A_a \in \alpha \circ \mu_c(P) - \alpha \circ I_c$ of P* wrt *μ_a and I_a, there is an inconsistency symptom $A_c \in \mu_c(P) - I_c$ of P* wrt *μ_c and I_c such that $\alpha(A_c) = A_a$.*
(2) *If $\alpha \circ I_c - \alpha \circ \mu_c(P) \neq \emptyset$, then P is insufficient* wrt *μ_c and I_c. For each insufficiency symptom $A_a \in \alpha \circ I_c - \alpha \circ \mu_c(P)$ of P* wrt *μ_a and I_a, there is an insufficiency symptom $A_c \in I_c - \mu_c(P)$ of P* wrt *μ_c and I_c such that $\alpha(A_c) = A_a$.*
(3) *If $\alpha \circ \mu_c(P) \neq \alpha \circ I_c$, then P is incorrect* wrt *μ_c and I_c.*

Proof. We prove (1) by contradiction. Assume P is consistent wrt μ_c and I_c. By definition 1, $\mu_c(P) - I_c = \emptyset$. $\alpha \circ \mu_c(P) - \alpha \circ I_c = \emptyset$. This is in contradiction with the premise of (1). Therefore, P is inconsistent wrt μ_c and I_c. This completes the proof of (1). (2) can be proved in a similar way and (3) results from (1) and (2). $\square$

By lemma 2, $\alpha \circ \mu_c(P) \neq \alpha \circ I_c$ implies that P is incorrect wrt μ_c and I_c. The comparison between $\alpha \circ \mu_c(P)$ and $\alpha \circ I_c$ is feasible because D_a is of finite size. However, there is no direct way of calculating $\alpha \circ \mu_c(P)$ or $\alpha \circ I_c$ in finite time unless both of $u_c(P)$ and $u_c(P_{spec})$ are finite. So, instead of comparing $\alpha \circ \mu_c(P)$ and $\alpha \circ I_c$, algorithm 1 compares $\mu_a(P)$ and I_a and uses the difference between them as a guide to the generation of test data.

A program is tested with respect to a given program property. To do this, algorithm 1 takes as an input the intended abstract semantics I_a corresponding to the program property. I_a is finite and can either be specified or be delivered by an abstract oracle. The abstract oracle is capable of deciding whether an element in $\mu_a(P)$ is in I_a and whether any element in I_a is missing from $\mu_a(P)$.

Algorithm 1. *Let μ_a be an abstract interpretation relative to μ_c and (α, γ).*

- *Inputs: P the program, I_a the intended abstract semantics and I_c the intended concrete semantics.*
- *Outputs: W_c a set of inconsistency symptoms of P* wrt *μ_c and I_c, and M_c a set of insufficiency symptoms of P* wrt *μ_c and I_c.*
- *Specifications for auxillary functions:*
  - *$\gamma'(A_a)$ terminates and $\gamma'(A_a)$ is a finite subset of $\gamma(A_a)$;*
  - *$solve/2$ terminates and $solve(A_c, P)$ implies $A_c \in \mu_c(P)$;*
  - *$verify/2$ terminates and $verify(A_c, I_c)$ implies $A_c \in I_c$.*
- *Algorithm:*

```
W_a := μ_a(P) − I_a
W_c := ∅
for each A_a ∈ W_a do
    T_{A_a} := γ'(A_a))
    for each A_c ∈ T_{A_a} do
        if solve(A_c, P) then W_c := W_c ∪ {A_c}

M_a := I_a − μ_a(P)
M_c := ∅
for each A_a ∈ M_a do
    T_{A_a} := γ'(A_a)
   for each A_c ∈ T_{A_a} do
        if verify(A_c, I_c) then M_c := M_c ∪ {A_c}

return W_c, M_c
```

Another input to algorithm 1 is the intended concrete semantics I_c. I_c is delivered by a concrete oracle. The concrete oracle is queried by auxiliary function $verify/2$ in order to decide whether an element in $\mu_c(P)$ is in I_c. The concrete oracle must declare $A_c \notin I_c$ for any A_c not in I_c whilst it may either declare $A_c \in I_c$ or declare $A_c \notin I_c$ for any A_c in I_c. If the concrete oracle makes a mistake by declaring an A_c in I_c to be not in I_c then this mistake will only reduce the number of the insufficiency symptoms of P *wrt* μ_c and I_c that algorithm 1 returns but will not compromise the soundness of algorithm 1. This tolerance is meant to ensure the termination of a mechanised concrete oracle.

Auxiliary function $solve/2$ emulates the concrete semantic function μ_c. The specification for $solve/2$ allows $solve(A_c, P)$ to evaluate into *false* when $A_c \in \mu_c(P)$. This is designed to ensure the termination of the emulation of the concrete semantic function μ_c.

Auxiliary function $\gamma'(A_a)$ generates test data by returning a finite subset of $\gamma(A_a)$. An implementation of γ' would be dependent on the program property against which the program is tested and need to adopt certain testing hypotheses such as the uniformity hypothesis and the regularity hypothesis [2, 1, 13].

Lemma 3. *Let μ_a be an abstract interpretation relative to μ_c and (α, γ), and W_c and M_c be results from algorithm 1.*

(1) If $A_c \in W_c$ then A_c is an inconsistency symptom of P wrt *μ_c and I_c.*
(2) If $A_c \in M_c$ then A_c is an insufficiency symptom of P wrt *μ_c and I_c.*

Proof. Consider (1). $A_c \in W_c$ implies $\alpha(A_c) \in \mu_a(P) - I_a$ by algorithm 1. So, $A_c \notin I_c$. We also know from algorithm 1 that $A_c \in \mu_c(P)$. So, A_c is an inconsistency symptom of P wrt μ_c and I_c by definition 1. This completes the proof of (1). (2) can be proved in a similar way. □

$solve/2$ is not used to detect insufficiency symptoms of P wrt μ_c and I_c because $A_c \in \gamma(I_a - \mu_a(P))$ implies $A_c \notin I_a$ and $verify/2$ is not used to detect inconsistency symptoms of P wrt μ_c and I_c because $A_c \in \gamma(\mu_a(P) - I_a)$ implies $A_c \notin I_c$. This typically results in a significant gain in efficiency and reduction in user interaction. The price for the gain in efficiency and the reduction in user interaction is that P is not tested on any input from $\gamma(\mu_a(P) \cap I_a)$. Because $\mu_a(P)$ and I_a are supersets of $\alpha \circ \mu_c(P)$ and $\alpha \circ I_c$ respectively, a discrepancy between $\mu_a(P)$ and I_a does not necessarily imply any discrepancy between $\mu_c(P)$ and I_c. In other words, that A_a is a bug symptom of P wrt μ_a and I_a does not imply that there is some $A_c \in \gamma(A_a)$ such that A_c is a bug symptom of P wrt μ_c and I_c. Algorithm 1 deals with this uncertainty by using $solve/2$ and $verify/2$.

Algorithm 1 uses the difference between $\mu_a(P)$ and I_a as a guide to test data generation. Algorithm 1 does not test P on any input from outside $\gamma(\mu_a(P) \cup I_a)$ because such an input cannot be a bug symptom of P wrt μ_c and I_c. Algorithm 1 does not test P on any input from $\gamma(\mu_a(P) \cap I_a)$ in order to reduce user interaction. Because there is a loss of information during abstract interpretation, there is probably some bug symptom $A_c \in \gamma(\mu_a(P) \cap I_a)$ of P wrt μ_c and I_c. So, algorithm 1 is incomplete. Incompleteness is not a weakness peculiar to algorithm 1. It has been proved that complete testing cannot in general be achieved [9, 17]. In our case, this incompleteness may be alleviated by testing P against multiple program properties.

4 Examples

Algorithm 1 applies to any logic programming language L, any concrete semantic function μ_c, any abstract semantic function μ_a, any abstraction function α and any concretization function γ as long as μ_a is an abstract interpretation of L relative to μ_c and (α, γ). We now illustrate algorithm 1 by taking as L the definite logic programming language and concrete semantic function μ_c of L as the least fixpoint semantic function in [21] which is restated here. Let P be a definite logic program, B_P the Herbrand base of P. Then $D_c = B_P$ and $\mu_c = \lambda P.lfp \circ T_P$ where $T_P : \wp(B_P) \mapsto \wp(B_P)$ is the following function.

$$T_P(I) = \left\{ A_0\sigma \middle| \begin{array}{l} \exists\sigma.\exists A_0 \leftarrow A_1, \ldots, A_m \in P. \\ (A_1\sigma \in I, \ldots, A_m\sigma \in I, A_0\sigma \in B_P) \end{array} \right\} \quad (1)$$

Let D_a be the abstract domain of interest and α be an abstraction function $\alpha : B_P \mapsto D_a$. Lu and Greenfield [12] formulate an abstract semantic function $\mu_a = \lambda\alpha.\lambda P.lfp \circ \Phi_{P,\alpha}$, where $\Phi_{P,\alpha}$ is defined below. $\mu_a = \lambda\alpha.\lambda P.lfp \circ \Phi_{P,\alpha}$ is an abstract interpretation relative to $\mu_c = \lambda P.lfp \circ T_P$ and $(\alpha, \gamma = \lambda A_a.\{A_c | \alpha(A_c) = A_a\})$.

$$\Phi_{P,\alpha}(I_a) = \left\{ \alpha(A_0\sigma) \middle| \begin{array}{l} \exists\sigma.\exists C = A_0 \leftarrow A_1, \ldots, A_m \in P. \\ (\alpha(A_1\sigma) \in I_a, \ldots, \alpha(A_m\sigma) \in I_a) \end{array} \right\} \tag{2}$$

We illustrate algorithm 1 through depth abstraction [18] and type abstraction [11, 8]. Examples 1 and 2 illustrate algorithm 1 through depth abstraction and use an executable specification P_{spec} for P. $solve/2$ is implemented as a depth-bound meta-interpreter for Prolog [8] and we let $I_a = \mu_a(P_{spec})$ and $verify(A_c, I_c) = solve(A_c, P_{spec})$ because P_{spec} is executable. γ' is implemented such that $\gamma'(A_a)$ replaces each occurrence of _ by a new variable.

A depth abstraction function partitions B_P into a finite number of equivalent classes. Two ground terms belong to the same class if and only if they are identical to a certain depth k. For example, $p(f(a), g(b))$ is equivalent to $p(f(b), g(a))$ to depth 2. Let d_k represent the depth k abstraction function. d_k replaces each sub-term of depth k with an _ where _ denotes any term. We have $d_2(p(f(a), g(b)) = d_2(p(f(b), g(a)) = p(f(\_), g(\_))$.

Example 1. Suppose we have the following specification and its implementation.

$$P_{spec} = \left\{ \begin{array}{l} p(a,b),\ p(X,Y) \leftarrow q(Y,X), \\ q(a,b),\ q(r(r(X)), s(s(Y))) \leftarrow q(X,Y) \end{array} \right\}$$

$$P1 = \{p(a,b),\ p(X,Y) \leftarrow q(X,Y),\ q(a,b),\ q(r(X), s(Y)) \leftarrow q(X,Y)\}$$

Let the abstract semantic function be $\mu_a = \lambda P.lfp \circ \Phi_{P,d_2}$ by setting $\alpha = d_2$. We have

$$I_a = \{p(a,b), q(a,b),\ p(b,a),\ q(r(\_), s(\_)),\ p(s(\_), r(\_))\}$$

$$\mu_a(P1) = \{p(a,b),\ q(a,b),\ q(r(\_), s(\_)),\ p(r(\_), s(\_))\}$$

and $I_a \neq \mu_a(P1)$. By applying algorithm 1, we have

$$W_a = \mu_a(P1) - I_a = \{p(r(\_), s(\_))\}$$

$$M_a = I_a - \mu_a(P1) = \{p(b,a),\ p(s(\_), r(\_))\}$$

and

$$W_c = \{p(r(a), s(b))\}$$

$$M_c = \{p(s(s(b)), r(r(a))),\ p(b,a)\}$$

The following example shows how the use of multiple program properties is useful in testing.

Example 2. Suppose we have the same specification as example 1 and we have the implementation that results from changing the positions of the arguments in the body goal $q(X,Y)$ of the second clause of $P1$ in example 1.

$$P2 = \{p(a,b),\ p(X,Y) \leftarrow q(Y,X),\ q(a,b),\ q(r(X),s(Y)) \leftarrow q(X,Y)\}$$

It can be shown that algorithm 1 does not find any inconsistency or insufficiency symptom when $\alpha = d_2$ because $I_a = \mu_a(P2)$. We now use another abstract semantic function $\mu_a = \lambda P.lfp \circ \Phi_{P,d_3}$ by setting $\alpha = d_3$. Then

$$I_a = \{p(a,b),\ q(a,b),\ p(b,a),\ q(r(r(\_)),s(s(\_))),\ p(s(s(\_)),r(r(\_)))\}$$

$$\mu_a(P2) = \left\{ \begin{array}{l} p(a,b),\ q(a,b),\ p(b,a),\ q(r(a),s(b)),\ p(s(b),r(a)), \\ q(r(r(\_)),s(s(\_))),\ p(s(s(\_)),r(r(\_))) \end{array} \right\}$$

$I_a \neq \mu_a(P2)$ and hence algorithm 1 is applicable. We have

$$W_a = \{q(r(a),s(b)),\ p(s(b),r(a))\}$$

$$M_a = \{\}$$

and

$$W_c = \{p(s(b),r(a)),\ q(r(a),s(b))\}$$

$$M_c = \{\}$$

The new program property, corresponding to $\alpha = d_3$, has helped to find two inconsistency symptoms of P wrt μ_c and I_c that are not found when the program is tested with respect to the program property corresponding to $\alpha = d_2$.

The above two examples have used an executable specification. The convenience of using an executable specification is that both the intended concrete semantics I_c and the intended abstract semantics I_a can be derived from the specification. However, the specification for a program is rarely given as an executable program. Instead, a number of program properties may be specified. In example 3, I_a is a type specification for P, γ' enumerates ground terms of a type using a depth-bound constraint, $solve/2$ is implemented as a depth-bound meta-interpreter and $verify/2$ is implemented by means of imposing queries on the programmer.

Example 3. Let $P = \{double(0,0), double(s(X),s(Y)) \leftarrow double(X,Y)\}$. P is a buggy implementation for the specification that $double(X,Y)$ is true if and only if $Y = 2 * X$. We follow Frühwirth in using a particular predicate ':/2' for type declarations [8]. The type declarations for the functors in P are

$$\{0 : even,\ s(X) : even \leftarrow X : odd,\ s(X) : odd \leftarrow X : even\}$$

$0 : even$ means that 0 is of type $even$. $s(X) : even \leftarrow X : odd$ means that if X is of type odd then $s(X)$ is of type $even$. A possible type specification for P is

$$I_a = \{double(odd,even),\ double(even,even)\}$$

Let $\mu_a = \lambda P.lfp \circ \Phi_{P,type}$ by setting $\alpha = type$ where $type(A_c) = A_a$ if and only if $A_c : A_a$ is true. The derived type for P is

$$\mu_a(P) = \{double(odd, odd), double(even, even)\}$$

$I_a \neq \mu_a(P)$ and hence algorithm 1 is applicable. We have

$$W_a = \{double(odd, odd)\}$$
$$M_a = \{double(odd, even)\}$$

and

$$W_c = \{double(s(0), s(0))\}$$
$$M_c = \{double(s(0), s(s(0)))\}$$

The following two queries are asked by $verify/2$:

```
Is double(s(0),0) valid? n.
Is double(s(0),s(s(0))) valid? y.
```

5 Adaptations

Examples 1 and 2 use an executable specification and example 3 uses a type specification and requires the programmer to verify if a ground atom A_c is valid in I_c or not. There are other cases that are of concern. In this section, we show how to adapt algorithm 1 to cater for these different cases.

5.1 Incomplete specifications

We first consider the case where we have an executable specification P_{spec} but P_{spec} does not cover all the procedures of P.

Examples 1 and 2 assume that each procedure of P is specified by P_{spec}. This is restrictive because this assumption requires that a decomposition of P_{spec} is available and is followed by the programmer, whilst P_{spec} should only be concerned with the functionality of P rather than the way that the functionality is to be attained. When the decomposition of P differs from that of P_{spec}, it is sensible to test only those procedures of P that are specified by P_{spec}. Algorithm 1 applies with minor modification: assign to W_a and M_a only those elements of D_a that correspond to the procedures of P specified by P_{spec}.

5.2 Correctness assertions

There are cases where some procedures of the program are known to be correct. For example, the program may contain some procedures from validated libraries or the programmer may have obtained sufficient confidence in some procedures during debugging. This kind of knowledge can be conveyed incrementally to algorithm 1 and be used to reduce the size of W_a and M_a. Algorithm 1 can be improved by deleting from W_a and M_a those elements of D_a that correspond to the procedures that are known to be correct.

6 Conclusion and further work

We have presented an algorithm (algorithm 1) that tests a program with respect to a given program property. The program properties appropriate to algorithm 1 are those derivable by means of abstract interpretation. Instead of comparing the actual concrete semantics and the intended concrete semantics of the program, algorithm 1 first compares the actual abstract semantics and the intended abstract semantics of the program and then uses the difference between them to guide test data generation. We have also shown how algorithm 1 can be adapted to cater for different cases.

Yan [22] studies declarative testing using a reliable test data set. What is not covered by Yan is how a reliable test data set is selected. Informally, a test set T is reliable for program P and specification P_{spec} if that P computes the same value as P_{spec} upon each point over T implies that P computes the same value as P_{spec} upon each point at which P is defined. A finite reliable test data set T for program P and specification P_{spec} exists, but there is no effective procedure to either generate or recognise a reliable test set [9, 17].

Dershowitz et. al [7] study logic program testing when an executable specification for the program is available. The program is exhaustively compared with the specification when a goal to be tested has been given. This is impractical because both the intended concrete semantics of the program specified by the specification and the actual concrete semantics of the program are usually infinite.

Logic program testing is an area which has rarely been looked at. There are only a few publications in this area. It does not appear that there has been any other work on using abstract interpretation in in logic program testing. Therefore, there are many topics to be addressed. Firstly, we need to find out what program properties are effective. We can test the program against as many program properties as possible. The more program properties against which the program is tested, the more confidence we have in the correctness of the program. However, there must be a trade-off between the number of the program properties against which the program is tested and the cost at which this testing is conducted. Therefore, we need to find out a set of program properties that are effective in exposing bugs. Secondly, we also need to know what program property is more useful than other program properties in exposing a particular kind of bug.

Acknowledgements

The first author is supported by the Sino-British Friendship Scholarship Scheme. Thanks are extended to Dr M. Ducassé for useful comments on an earlier version of this paper.

References

1. G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. LRI research report no. 581, April 1991.
2. L. Bouge, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generating from algebraic specifications using Logic Programming. *Journal of System and Software*, 6(4):343–360, 1986.
3. M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 669–683. The MIT Press, 1988.
4. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimisation of prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. The IEEE Society Press, 1987.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on Principles of programming languages*, pages 238–252, Los Angeles, California, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the sixth annual ACM symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, 1979.
7. N. Dershowitz and Y.-J. Lee. Deductive debugging. In *Proceedings of 1987 Symposium of Logic Programming*, pages 298–306. The IEEE Computer Society Press, 1987.
8. T.W. Früehwirth. Using meta-interpreters for polymorphic type checking. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 339–351, Leuven, Belgium, April 4-6 1990.
9. W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, 1976.
10. N.D. Jones and H. Søndergarrd. A semantics-based framework for abstract interpretation of prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, 1987.
11. T. Kanamori and K. Horiuchi. Type inference in prolog and its application. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 704–707, 1985.
12. L. Lu and P. Greenfield. Abstract fixpoint semantics and abstract procedural semantics of definite logic programs. In *Proceedings of IEEE Computer Society 1992 International Conference on Computer Languages*, pages 147–154, Oakland, California, USA, April 20-23 1992. IEEE Computer Society Press.
13. B. Marre. Toward automatic test data set selection using algebraic specifications and Logic Programming. In K. Furukawa, editor, *Proc. of the Eighth International Conference on Logic Programming*, pages 202–219, Paris, France, 1991. ALP, MIT Press.
14. K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 733–748. The MIT Press, 1988.
15. C. Mellish. Some global optimisations for a prolog compiler. *The Journal of Logic Programming*, 2(1):43–66, 1985.

16. C. Mellish. Abstract interpretation of prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 181–198. Ellis Horwood Limited, 1987.
17. L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.
18. T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34(1):227–240, 1984.
19. D.A. Schmidt. *Denotational Semantics: a methodology for language development*. Allyn and Bacon, Inc., 1986.
20. H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. In **LNCS 213** *Proceedings of the European symposium on programming*. Springer-Verlag, 1986.
21. M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Artificial Intelligence*, 23(10):733–742, 1976.
22. S.Y. Yan. Declarative testing in logic programming. In *Proceedings of the Third Australian software engineering conference*, pages 423–435, 1987.

This article was processed using the LaTeX macro package with LLNCS style