This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Jack, O., A Test Coverage Notion for Logic Programming, Proc. 6th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp.133-142

A Test Coverage Notion for Logic Programming

Fevzi Belli Oliver Jack
University of Paderborn, Dept. of Electrical and Electronics Engineering,
33095 Paderborn, Germany
Email: belli@adt.uni-paderborn.de oliver@adt.uni-paderborn.de

Abstract

Reliability determination for software is closely related to software testing. Testing delivers important data for software reliability models. Two important tasks of software testing are test case generation and determination of test coverage. Because of its declarative paradigm, all the well-known approaches to the above mentioned tasks are not applicable to logic programming. Implementation-based testing is an approach to tackle the test problem for logic programming [1, 2].

In the present paper, we focus on test coverage aspects for logic programming. Analog to the coverage notion for conventional programming, e.g., path coverage on the control flow graph, we define a coverage measure on an abstract model of the logic program to be tested. Our abstract model is the set of goal-induced instances *of program clauses. This is motivated by the computational model of resolution calculus for first-order predicate logic.* Anti-unification *is utilized to define the coverage measure via least upper bounds of this set. We give an algorithm for test cover determination which is prepared for* coverage-oriented test input generation, *e.g., by declarative program instrumentation or non goal-oriented execution of logic programs using a fair interpreter.*

The paper describes syntactic and semantic aspects of our testing approach with respect to the properties of logic programming.

Keywords: Software testing, logic programming, test coverage measures, anti-unification, coverage-oriented test input generation.

1 Introduction

Program testing must be evaluable in the sense that a measure must be available to express to what extent which properties of a program under test have been investigated by a set of test inputs. Such a measure is called a coverage measure. Coverage is not measured on the program itself but on an abstract model of the program expressing the program properties to be investigated. Common coverage measures are defined on the control flow or data flow graph of a program [11, 12]. This is justified in case of programs implemented in an imperative programming language since such programs explicitly express the control and data flow. The results computed by such measures are reasonable since they are directly related to the program code. Coverage-oriented testing deals with test inputs only instead of test cases. A *test input* is an item suitable for activation of a program, i.e., an input to the program to be tested which meets the syntactical requirements for executing a function of the program. A *test case* is a pair of a test input and its expected output. Test cases are used to compare the actual output with the expected output (black-box testing). Coverage does only consider the implementation and not the specification (white-box testing).

In some more recent programming paradigms, such as logic programming, the program consists of assertions in form of facts and rules about a problem and it is up to the system to figure out the solution. Hence no control flow is encoded in a logic program and consequently no control flow graph can be extracted. Classical coverage measures are therefore not applicable [6]. The problem arises to define an abstract model of a logic program which is suitable to define a coverage measure upon. The choice of such a model and measure must meet the following requirements to ensure expressiveness with respect to test coverage:

1. the empty test input set must have coverage 0,
2. the coverage measure c must be monotone, i.e., if T, T' are test input sets for program P with $T \subseteq T'$ then $c(T, P) \leq c(T', P)$,
3. exhaustive testing always covers a program.

In the next section, we analyze the computational properties of logic programming and develop a test coverage notion which meets the above requirements.

2 Preliminaries

In this section we summarize the notions to be used in throughout the paper, without being complete, i.e., only to explain our test approach.

Logic programming is based on first-order predicate logic. More precisely, a logic program consists of a finite set of *definite Horn clauses*. Horn clauses are universal quantified disjunctive clauses with at most one positive literal. These are classified in three types called *fact*, *rule*, and *goal*. Definite clauses, or program clauses, are facts and rules. A rule has a head A and body $B_1, \ldots, B_n$. As usual, we denote Horn clauses by

$$\begin{array}{ll} A, & \text{(fact)} \\ A \leftarrow B_1, \ldots, B_n, & \text{(rule)} \\ \leftarrow B_1, \ldots, B_n. & \text{(goal)} \end{array}$$

A fact states something which is true, a rule has the meaning that its head A is true provided that the conjunction of its body literals $B_1, \ldots, B_n$ is true. A goal is used to activate a logic program P, i.e., it is a question if the conjunction of the litarals $B_1, \ldots, B_n$ follows from P.

The first-order predicate logic language (we skip the prefix "predicate logic" in the following, and call such languages "first-order") underlying a program is denoted by $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ where $\mathcal{R}$ is the set of predicate symbols, $\mathcal{F}$ is the set of function symbols, and $\mathcal{V}$ is the set of variable symbols. The sets $\mathcal{R}$ and $\mathcal{F}$ are divided into sets $\mathcal{R}_n$ and $\mathcal{F}_n$ ($n \in \mathbf{N}$), denoting the subsets of $\mathcal{R}$ and $\mathcal{F}$ of symbols with arity n. The set of terms is denoted by $\mathcal{L}(\mathcal{F}, \mathcal{V})$, the variable-free terms are denoted by $\mathcal{L}(\mathcal{F})$, and the variable-free clauses are denoted by $\mathcal{L}(\mathcal{R}, \mathcal{F})$.

The central syntactical manipulation framework of first-order predicate logic, the *Substitution* is a finite mapping $\sigma: \mathcal{V} \rightarrow \mathcal{L}(\mathcal{F}, \mathcal{V})$ from the set of variables to the set of terms, which is identified with a finite set $\{X_1/t_1, \ldots, X_m/t_m\}$ of ordered pairs X_i/t_i where X_i are distinct variables and t_i are terms and each t_i is distinct from X_i, $(1 \leq i \leq m)$. An ordered pair X/t is called a *binding* of X to t. Substitutions are denoted post-fix, $t\sigma := \sigma(t)$. The application $t\sigma$ of σ to a term t is the term t' obtained by simultaneously replacing each occurrence of a variable X where $X/s \in \sigma$ by s. The term t' is called the *instance* of t by σ. An instance is *ground* if there are no occurrences of variables in it. The set of all ground instances of a term t is denoted by $\mathcal{G}(t)$. The above instance definition extend to sets of terms, clauses and sets of clauses in a canonical way.

We denote the instance relation by $\sqsubseteq$, i.e., $t \sqsubseteq s$ if t is an instance of s. By $\sqsubseteq$, the set of terms and the set of clauses can be factored into equivalence classes. Two terms t_1, t_2 or clauses C_1, C_2 are *variants* if $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_1$, resectively $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. The variant relation obviously is an equivalence relation, it is denoted by $\simeq$. A set M of terms or clauses is *unifiable* if there exists a substitution σ such that $M\sigma$ is a singleton. A *most general unifier* (mgu), which exists for any unifiable set M, is a substitution μ such that for any other unifier σ for M, there exists a substitution θ with $\sigma = \theta \circ \mu$. Compositions $\theta \circ \sigma$ of substitutions are denoted $\sigma\theta := \theta \circ \sigma$.

For the semantics, we consider the *Herbrand universe* $\mathcal{H}_{\mathcal{L}} := \mathcal{L}(\mathcal{F})$ and *Herbrand base* $B_{\mathcal{L}} := \mathcal{L}(\mathcal{R}, \mathcal{F})$ of a program. We assume that the first-order language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ for a program P is given by the symbols occurring in the program clauses of P. Hence we write $\mathcal{H}_P$ for the Herbrand universe and B_P for the Herbrand base of a program. The semantics of a program P is the set $\mathcal{M}^+(P)$ of ground atomic formulae which are logical consequences of P. $\mathcal{M}^+(P)$ is called the *minimal Herbrand model* of P.

Procedurally, goals are used to determine if a formula is a logical consequence of a program. The computational model is *resolution calculus*, which consists of one single rule. The inference rule is

$$\frac{\{L\} \cup C_1 \quad \{\sim L\} \cup C_2}{(C_1 \cup C_2)\mu},$$

where C_1, C_2 are Horn clauses, L is a literal, and μ is a mgu of $\{L\} \cup C_1$ and $\{\sim L\} \cup C_2$. For a literal L, the *negation* of L is denoted by $\sim L$, i.e., if L is an atomic formula A, then $\sim L = \neg A$, and if L is a negated atomic formula $\neg A$, then $\sim L = A$. Note that we always use a goal and a program clause for resolution. Because goals have only negative literals and a program clause has exactly one positive literal, the only choice in selecting a literal for building a resolvent is among the literals of the goal. Hence, we always unify the head of a definite clause (its only positive literal) with an atomic formula in a goal. How we select this literal from the possible ones is called a *computation rule*. In logic programming we always select the first (leftmost) literal. This kind of resolution is called *SLD-resolution*.

SLD stands for *Selective Linear Derivation*.

Using the resolution calculus, we try a refutation of $P \cup \{\leftarrow G\}$ in order to compute an answer. We start with a goal clause G and choose some appropriate variant C_0 of a program clause in order to build a resolvent $G_1\mu_1$ of G and C_0 if such a C_0 exists. Then we proceed in the same way with the goal $G_1\mu$, producing a sequence

$$\langle (G_0, \mu_0), (G_1, \mu_1), \ldots, (G_{n-1}, \mu_{n-1}), (G_n, \mu_n) \rangle,$$

where each (G_i, μ_i) pair is an *immediate derivant* of goal G_{i-1} $(0 \leq i \leq n)$, i.e., there exists some literal $L \in G_{i-1}$ and a (suitable variant of) a program clause $C = (H \leftarrow B) \in P$ such that the *mgu* of L and H is μ_i and $G_i = ((G_{i-1} \setminus \{L\} \cup B)\mu_i$. There are three possibilities. If $G_n = \Box$ then the derivation is *successful* and the substitution $\mu_0 \cdots \mu_n$ is a *computed answer* for $P \cup G$. In this case $G'\mu_1 \cdots \mu_n$ is a logical consequence of P for $G = \leftarrow G'$. To be precise, the substitution $\mu_0 \cdots \mu_n$ is restricted to the variables occurring in G. If $G_m \neq \Box$ and there is no clause in P to build a resolvent, the derivation is *failed*. The process does not terminate and the derivation is *infinite*.

The search strategy is to arrange the derivations for $P \cup \{G\}$ in form of a tree, called an *SLD-tree* which is searched depth-first left-to-right for successful derivations. This method is correct and complete with respect to logical consequences from P. The set of all G such that $P \cup \{\leftarrow G\}$ has a successful derivation is called the *success set* of P and denoted by $\mathcal{S}^+(P)$. Obviously $\mathcal{M}^+(P) \subseteq \mathcal{S}^+(P)$ and $\mathcal{G}(\mathcal{S}^+(P)) = \mathcal{M}^+(P)$.

3 Anti-Unification

In logic programming, unification is used to extract specific information out of a program which asserts general information about a problem. In particular, the program clauses are instantiated via goals, leading to specific instances of these clauses. Repeating this process, concrete information is extracted from a program. A test coverage notion for logic programs must consider this computational model. We build our coverage notion on the abstract model of the program instances induced by goals and resolution. For this, we define a relation on the set of program instances. The concept behind is *anti-unification*.

Because anti-unification is not broadly known, we give an adoption of its main topics, suited for our needs. The full theory can be found in e.g., [8]. For technical reasons, we restrict to terms, but all the results can be applied to Horn clauses as well. This can be done if we view a Horn clause $H \leftarrow B$ as a term with 2-ary (infix) function symbol '$\leftarrow$' having the arguments H and B. The body $B = (B_1, \ldots, B_n)$ can be seen as a nested term with 2-ary (infix) function symbol ','.

$$B = (B_1, \ldots, B_n) = (B_1, (B_2, \cdots (B_{n-1}, B_n) \cdots).$$

We extend the set of terms by a least element $\perp$. Now, the instance relation is a partial ordering on the set of terms. Most general unification of a (unifiable) set M of terms corresponds to the greatest lower bound of M under $\sqsubseteq$, i.e., if μ is an mgu for M then $M\mu$ is the greatest lower bound of M. In case M is not unifiable, we define $\perp$ as the greatest lower bound of M.

Next, we investigate the least upper bound of a set of terms.

Definition 1 *Let M be a nonempty set of terms. A term t is an* anti-instance *of M if for all $s \in M, t \sqsupseteq s$. A term t is a* least common anti-instance *of M, denoted $t = \mathrm{lca}(M)$, if*

1. *t is an anti-instance of M*
2. *for each anti-instance t' of M, $t' \sqsupseteq t$.*

From the definition of lca it follows that all lca of a set M of terms are instances of each other. Thus an lca is unique modulo renaming.

The concept of the least common anti-instance is inspired by the unification concept of logic programming: unification leads to the greatest lower bound of terms under the instance ordering, lca are least upper bounds. For this reason, the computation of an *lca* is called anti-unification.

Example 2 Let $M = \{\mathrm{p(1, X, [a|R]), p(Y, 2, [a,b|S])}\}$. Then $\mathrm{p(V, W, [a|R])}$ is an lca of M. Let $N = \{\mathrm{p(1, X, [a|R]), q(1, X, [a|R])}\}$. Then W is an lca of N. In this case every common anti-instance of N is a least common anti-instance of N. The connection between unification and anti-unification is depicted in Figure 1.

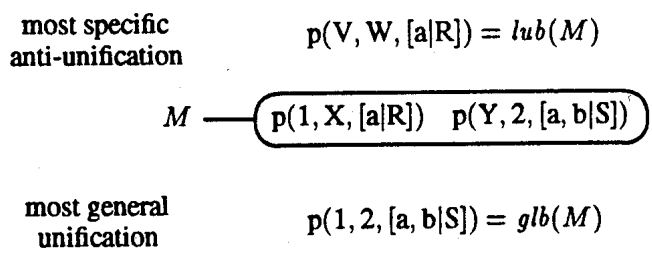

Figure 1: Unification and anti-unification

Algorithm 1 (Anti-unification)

```
function antiunify(t, s): a;
input:   t, s: terms
output:  a: term
begin
  if (t = f(t_1, ..., f_n) and s = f(s_1, ..., s_n)) then
    begin
      for i := 1 to n do
        a_i := antiunify(t_i, s_i);
      a := f(a_1, ..., a_n);
    end
  else a := φ(t, s);
end.
```

An important result is the following theorem of which the proof can be found in [8]:

Theorem 3 (Anti-unification) *Let M be a nonempty set of terms. Then M has an* lca *which is unique modulo variants.*

The Anti-unification Theorem states that the set of terms, extended by $\perp$, together with the instance relation, forms a complete lattice. For each nonempty set of terms, there exist a least upper bound (by the Anti-unification Theorem), and a greatest lower bound (since either the set is unifiable and the mgu is the greatest lower bound or the greatest lower bound is $\perp$).

Now that we know that any set of terms has an lca we consider the question how to calculate the lca. For this we consider the anti-unification algorithm which returns an lca on input of two terms. The simplest algorithm is from [7], it is a nondeterministic algorithm. Let

$$\phi: \mathcal{T}_{\mathcal{F},\mathcal{V}} \times \mathcal{T}_{\mathcal{F},\mathcal{V}} \rightarrow \mathcal{V}$$

be a bijection. The Anti-unification Algorithm is defined recursively on $\mathcal{T}_{\mathcal{F},\mathcal{V}} \times \mathcal{T}_{\mathcal{F},\mathcal{V}}$. The lca computed by this algorithm depends on the choice of ϕ. But for any choice of ϕ and ϕ', the lca are equivalent, i.e., if a is computed using some ϕ and a' using ϕ' then $a \simeq a'$.

Example 4 Let $t = f(X, Y, g(a, X))$ and $s = f(a, g(W, Z), g(a, b))$. Then

$$antiunify(t, s) = f(X, Y, g(a, Z))$$

with $\phi(X, a) = X, \phi(Y, g(W, Z)) = Y$ and $\phi(X, b) = Z$.

Note that the case $t = \perp$ or $s = \perp$ is not considered. This could be done by spawning the **else**-part in Algorithm 1 into

```
elsif t = ⊥ then
  a := s
elsif s = ⊥ then
  a := t
else a := φ(t, s)
```

Algorithm 1 describes the computation of the lca of two terms. This algorithm can also be used to compute the lca of an arbitrary finite set of terms. The reason is that the lca can be defined as a 2-ary mapping

$$\text{lca}: \mathcal{L}(\mathcal{F}, \mathcal{V}) \times \mathcal{L}(\mathcal{F}, \mathcal{V}) \rightarrow \mathcal{L}(\mathcal{F}, \mathcal{V}),\ (t, s) \mapsto lub(\{t, s\}),$$

which is associative (and, of course, commutative) [8], i.e., for any expressions t_1, t_2, and t_3

$$\begin{aligned} \text{lca}(\{t_1, t_2, t_3\}) &\simeq \text{lca}(\{\text{lca}(\{t_1, t_2\}), t_3\}) \\ &\simeq \text{lca}(\{t_1, \{\text{lca}(\{t_2, t_3\})\}). \end{aligned}$$

This enables the computation of the lca of a finite set M of terms by successively computing the lca of the elements of M in any order.

4 Coverage and Anti-Unification

Whereas unification is used in logic programming to gain specific information out of a general program, anti-unification will be used to gain general information out of specific goals, i.e., test inputs. In the operational semantics of a logic program P, a goal $\leftarrow G$ initiates a refutation attempt starting with the unification of G with the head H of an appropriate clause $C = (H \leftarrow B)$ of P. This leads to an instance $(H \leftarrow B)\mu$ of C, where μ is an mgu of G and H. The body instance $B\mu$ is the goal reduction for further derivation of $P \cup \{\leftarrow G\}$ (cf. section 2). Hence, covering a logic program means to cover the goal reductions which stem from clause instances $(H \leftarrow B)\mu$. The general information which can be gained from a set CI of clause instances, derived from a set of goals, is the least common anti-instance of CI. The lca of CI tells us about how the test inputs cover the possible clause instances of the program. This gives the clue to the coverage notion for logic programs, which we will call cover (cf. [3]).

Definition 5 *Let $T = \{\leftarrow G_1, \ldots, \leftarrow G_m\}$ be a finite set of atomic goals. Let $H \leftarrow B$ be a program clause. The* instantiation set *of $H \leftarrow B$ by T is*

$$\begin{aligned} &\{\!\!\{H \leftarrow B\}\!\!\}_T := \\ &\quad \{\, (H \leftarrow B)\mu : \exists \leftarrow G \in T \textit{ such that} \\ &\quad\quad G \textit{ and } H \textit{ are unifiable with mgu } \mu \,\}. \end{aligned}$$

Let $P = \{C_1, \ldots, C_k\}$ be a program. The instantiation set *of P by T is*

$$\{\!\![P]\!\!\}_T := \bigcup_{i \in \{1,\ldots,k\}} \{\!\![C_i]\!\!\}_T.$$

The instantiation set has the desired properties which are required for a coverage notion.

Lemma 6 *For each finite set of atomic goals T and each program clause $C = H \leftarrow B$, either $\{\!\![C]\!\!\}_T = \emptyset$ or* $\mathrm{lca}(\{\!\![C]\!\!\}_T) \sqsubseteq C$.

Definition 7 *Let $T := \{\leftarrow G_1, \ldots, \leftarrow G_m\}$ be a finite set of atomic goals. Let $R := \{C_1, \ldots, C_q\}$ be the definition of the predicate p/n, in a program P.*

- *T is a* cover for a program clause *C_i, or T covers C_i, if*
  1. $\{\!\![C_i]\!\!\}_T \neq \emptyset$
  2. $\mathrm{lca}(\{\!\![C_i]\!\!\}_T) \simeq C_i$.
- *T is a* cover for the predicate *p/n, or T covers p/n, if T is a cover for each $C \in R$.*
- *T is a* cover for a program *P, or T covers P, if T is a cover for each predicate in P.*

The above coverage notion defines a structural coverage, but it is not based on control flow [14]. It is more related to data flow, since instances of clauses, i.e., substitutions of variables by terms, are central to our coverage notion. It also reflects the logical structure of the program P to be tested, since by clause instances its Herbrand base B_P is considered (in the case of ground instances). For illustration, consider the following examples.

Example 8 Let P be the following program:

C_1 : is_list([]).
C_2 : is_list([_|Tail]) ← is_list(Tail).

Let

$$T = \{\leftarrow \text{is\_list([])}, \leftarrow \text{is\_list([a])}, \leftarrow \text{is\_list([a,b])}\}.$$

The sets of top level instances of P are then

$$\begin{aligned} \{\!\![C_1]\!\!\}_T &= \{\text{is\_list([])}\}, \\ \{\!\![C_2]\!\!\}_T &= \{\text{is\_list([a])} \leftarrow \text{is\_list([])}; \\ &\quad \text{is\_list([a,b])} \leftarrow \text{is\_list([b])}\}. \end{aligned}$$

The least common anti-instances are

$$\begin{aligned} \mathrm{lca}(\{\!\![C_1]\!\!\}_T) &= \text{is\_list([])} \\ \mathrm{lca}(\{\!\![C_2]\!\!\}_T) &= \text{is\_list([a|Tail])} \leftarrow \text{is\_list(Tail)}. \end{aligned}$$

So the clause C_1 of P is covered, but clause C_2 not.

Example 9 Let P be the program

C_1 : append([], L, L).
C_2: append([A|T1], L, [A|T2]) ← append(T1, L, T2).

Let

$$T_1 = \{\leftarrow \text{append([], [], [])}, \leftarrow \text{append([], a, a)}\}$$

and

$$\begin{aligned} T_2 = \{ &\leftarrow \text{append([1], [], [1])}, \\ &\leftarrow \text{append([1,2], [], [1,2])}, \\ &\leftarrow \text{append([2], [1], [2,1])}\} \end{aligned}$$

be a sets of test inputs. Then T_1 is a cover for clause C_1 of P and T_2 is a cover for clause C_2 of P. Hence, $T_1 \cup T_2$ is a cover for P.

The above example shows that the notion of a coverage delivers more information than just determining whether a set of goals covers a program clause or not. The lca of the set of top level instances of the clause

is_list([_|Tail]) ← is_list(Tail).

was

is_list([a|Tail]) ← is_list(Tail).

This is an information about the *instance* of coverage of a clause by a set of test inputs. The following Definition states this formally.

Definition 10 *Let C be a program clause and T be a finite set of goals. T is a cover for the instance C' of C, or T covers the instance C' of C if the* lca *of T is a variant of C'. The instance C' of C covered by T is called the* coverage *of C by T. The coverage of a program P by T is the set of covered instances of the clauses of P.*

Definition 10 essentially means that if T is a cover for an instance C' of a clause C, where $C' \sqsubseteq C$ and $C' \neq C$, then, loosely speaking, T covers C *up to* C'. From the instance C' of C covered by T information can be gained for extending the instance covered. Here extending the instance is subject to the instance ordering $\sqsubseteq$. In Example 8 the instance

is_list([a|Tail]) ← is_list(Tail).

of clause

is_list([_|Tail]) ← is_list(Tail).

was covered. From the definition of the lca, it can be seen that choosing any additional test input

$\leftarrow$ is_list([x|*tail*]),

where x is different from the constant a and *tail* is any term, will lead to a cover of

is_list([_|Tail]) $\leftarrow$ is_list(Tail).

For the cover notion to be meaningful, it is required that the extension of a set T of goals does not decrease the instances of the clauses covered by T.

Lemma 11 *Let $P = \{C_1, \ldots, C_m\}$ be a program, T_0 be a finite set of atomic goals, and*

$$Cov_0 := \{\, \mathrm{lca}(\{\!\![(C_i)]\!\!\}_{T_0}) : (1 \leq i \leq m) \,\}.$$

Let $\leftarrow G$ be any atomic goal, $T_1 := T_0 \cup \{\leftarrow G\}$ and

$$Cov_1 := \{\, \mathrm{lca}(\{\!\![(C_i)]\!\!\}_{T_1}) : (1 \leq i \leq m) \,\}.$$

Then we have

$$\mathrm{lca}(\{\!\![(C_i)]\!\!\}_{T_0}) \sqsubseteq \mathrm{lca}(\{\!\![(C_i)]\!\!\}_{T_1}) \ : (1 \leq i \leq n).$$

Lemma 11 can be proved as follows. From Definition 1 follows that $\{\!\![C_i]\!\!\}_{T_0} \subseteq \{\!\![C_i]\!\!\}_{T_1}$ $(1 \leq i \leq m)$. Hence,

$$\mathrm{lca}(\{\!\![C_i]\!\!\}_{T_0}) \sqsubseteq \mathrm{lca}(\{\!\![C_i]\!\!\}_{T_1}) \ (1 \leq i \leq m).$$

Example 12 Consider again the append/3 program P.

C_1 : append([], L, L).
C_2 : append([A|T1], L, [A|T2]) $\leftarrow$ append(T1, L, T2).

Let T_0 be a set of test inputs

$$\{\leftarrow \mathrm{append}([], [], []), \leftarrow \mathrm{append}([1,2], [], [1,2])\}.$$

T_0 covers the instance

$$I_1 = \mathrm{append}([], [], []) = \mathrm{lca}(\{\!\![C_1]\!\!\}_{T_0}).$$

of clause C_1 and

$$\begin{aligned} I_2 &= \mathrm{append}([1,2], [], [1,2]) \leftarrow \mathrm{append}([2], [], [2]) \\ &= \mathrm{lca}(\{\!\![C_2]\!\!\}_{T_0}). \end{aligned}$$

of clause C_2. Extending T_0 by the test input $t_1 = \leftarrow$ append([], a, a) to get $T_1 = T_0 \cup \{t_1\}$ leads to a cover of the instance

$$I_3 = \mathrm{append}([], \mathrm{A}, \mathrm{A}) = \mathrm{lca}(\{\!\![C_1]\!\!\}_{T_1}).$$

of clause C_1 and the same instance I_2 of clause C_2 as before. Clearly, $I_2 \sqsubseteq I_2$. We have $I_1 \sqsubseteq I_3$. Further, we have $I_3 \simeq C_1$, hence clause C_1 is covered. Extending T_1 by the test input $t_2 = \leftarrow$ append([a], [a], [a,a]) to get $T_2 = T_1 \cup \{t_2\}$ leads to a cover of the instance

$$\begin{aligned} I_4 &= \mathrm{append}([\mathrm{A}|\mathrm{B}], \mathrm{C}, [\mathrm{A},\mathrm{D}]) \leftarrow \\ &\qquad \mathrm{append}(\mathrm{B}, \mathrm{C}, [\mathrm{D}]) \\ &= \mathrm{lca}(\{\!\![C_2]\!\!\}_{T_2}). \end{aligned}$$

of clause C_2 and the the same instance I_3 of clause C_1 as before. We have $I_2 \sqsubseteq I_4$ but $C_2 \not\sqsubseteq I_4$, hence clause C_2 is not covered. Extending the current test input set T_2 by the test input $t_3 = \leftarrow$ append([2,3], [4], [2,3,4]) to get $T_3 = T_2 \cup \{t_3\}$ leads to a cover of the instance

$$\begin{aligned} I_5 &= \mathrm{append}([\mathrm{A}|\mathrm{B}], \mathrm{C}, [\mathrm{A},\mathrm{D}|\mathrm{E}]) \leftarrow \\ &\qquad \mathrm{append}(\mathrm{B}, \mathrm{C}, [\mathrm{D}|\mathrm{E}]) \\ &= \mathrm{lca}(\{\!\![C_2]\!\!\}_{T_3}). \end{aligned}$$

of clause C_2. Again, we have $I_4 \sqsubseteq I_5$ but $C_2 \not\sqsubseteq I_5$, hence clause C_2 is still not covered.

As the above Example demonstrates, finding a *meaningful* cover, is not trivial. However, for any program there is always a *trivial* cover. Let $P = \{C_1, \ldots, C_m\}$ be a program consisting of clauses $C_1, \ldots, C_m$. Let $C_i = H_i \leftarrow B_i$ $(i \leq i \leq m)$. Then

$$\{\leftarrow H_1, \ldots, \leftarrow H_m\}$$

is a cover for P. A trivial cover, thus, consists of the heads of the program clauses. This is obviously a cover, since for any clause, the instantiation set $\{\!\![\{H \leftarrow B\}]\!\!\}_{\{\leftarrow H\}}$ includes a variant of $H \leftarrow B$.

A trivial cover for the append/3 program, e.g., is

$$\{\leftarrow \mathrm{append}([], \mathrm{L}, \mathrm{L}), \leftarrow \mathrm{append}([\mathrm{A}|\mathrm{T1}], \mathrm{L}, [\mathrm{A}|\mathrm{T2}])\}.$$

Of course, a trivial cover is not meaningful in the sense that it provides no additional information to the program.

A test input set for a program P is called *non trivial* if it contains only goals $\leftarrow G$ such that there exists no clause $H \leftarrow B$ in P with $G \not\simeq H$. A goal from a non trivial test input set for P is called a non trivial test input for P. As a consequence, non trivial test inputs for predicates consisting only of clauses with ground heads, must be non ground. In many cases, program clauses with ground heads are used to represent *explicit* data, e.g., facts of the form arc(a, b) representing an edge in a directed graph. "Testing" such explicit data is often not useful.

Program clauses with non ground heads are interesting for testing, since they contain general knowledge about the relation they encode. A special case of a non trivial test input for non ground clauses are ground goals. Next

the question if there always exists a cover of ground goals for an arbitrary program is investigated. Fortunately this question can be answered with yes, provided that a certain (reasonable) condition holds. We can prove the following theorem.

Theorem 13 *Let P be a program, such that its Herbrand universe $\mathcal{H}_P$ contains at least one constant and one n-ary function symbol ($n > 0$). Then for P there exists a cover T such that T contains only ground goals.*

Lemma 11 shows, that the cover notions 7 and 10 are exactly what we expect from a cover notion. Definition 7 provides the general notion of test coverage for logic programs. It defines the *qualitative* coverage notion in the sense of a decidable relation on the set of programs and test input sets. Definition 10 gives the *quantitative* coverage notion in the sense of defining the extent of testing with respect to the instance ordering $\sqsubseteq$ on the set of program clauses. Theorem 13 shows that any program indeed possesses a cover. But Theorem 13 provides no feasible constructive method for the generation of such a cover. It essentially states that *exhaustive testing* provides a cover.

Defining an instance ordering on the set of terms and considering least upper bounds of sets of terms under the instance ordering has also been used for optimization of logic programs [10]. The approach was to generate a *most specific version* of a logic program. This is obtained by replacing the clauses C of a program P by specific instances C', such that the resulting program P' has the same success set as P, i.e., $\mathcal{S}^+(P) = \mathcal{S}^+(P')$, but a possibly increased set of finitely failed goals. Hence P' can be more efficient than P since non-successful derivations may be detected more quickly.

In the next section, implementation aspects of coverage determination are discussed. We also consider test input generation aspects and incorporate an interface for coverage-oriented test input generation into the algorithm for test coverage computation.

5 Coverage Computation

In this section, the entire algorithm for the determination of the coverage of a set of test inputs to a program will be given. From the previous section, the obvious way to implement coverage computation, is Algorithm 2.

However, from the practical view, it is more convenient to implement coverage determination step by step for each test input. The reason is, that in this case, the coverage determination algorithm can be used for incremental test cover *generation* in a more efficient way than Algorithm 2. The algorithm will be given in the form of Prolog.

Algorithm 2 (Test cover determination)

function *coverage*$(T, P) : PC$;
input: T: set of goals
P: program, $P = (C_1, \ldots, C_m)$
output: PC: set of covered instances of the clauses in P
begin
$PC := \emptyset$;
for $i := 1$ **to** m **do**
$PC := PC \cup \mathrm{lca}(\{\!\{C_i\}\!\}_T)$
end.

Program 1 (Update cover)

```
update_cover(CPs0, Goal, CPs) ←
  ( select(ci(C, CI), CPs0, CPs1),
    copy_term(C, (CH ← CB)),
    copy_term(Goal, Goal1),
    CH = Goal1 →
    ( CI = [] →
      CI1 = (CH ← CB)
    ; anti_unify((CH ← CB), CI, CI1)
    ),
    ( subsumes_chk(CI1, C) →
      CPs = [c(C)|CPs2]
    ; CPs = [ci(C, CI1)|CPs2]
    ),
    update_cover(CPs1, Goal, CPs2)
  ; CPs = CPs0
  ).
```

The kernel of the program is the predicate update_cover/3, shown in Program 1, which defines a relation on the following sets:

- lists $[X_1, \ldots, X_m]$, where

$$X_i = \begin{cases} \mathrm{c}(C) & : \text{clause } C \text{ is covered} \\ \mathrm{ci}(C, CI) & : \text{instance } CI \text{ of } C \text{ is covered} \end{cases}$$

  and C is a clause, CI is either a an instance of C or [],

- terms, interpreted as goals.

In the program for coverage computation, we use the following data structures: *coverage* is a term c(*clause*) or

ci(*clause instance*) where *clause* is a term $A \leftarrow B_1, \ldots, B_n$ and *clause instance* is either a *clause* or [], indicating that there is no instance for that clause. The relation update_cover(CPs0, Goal, CPs) is true if CPs0 is a list of *coverage*, Goal is a term, and CPs is a list of *coverage* such that for each c(C) $\in$ CPs0 c(C) $\in$ CPs and for each ci(C, CI) $\in$ CPs0

- c(C) $\in$ CPs in case lca($CI, \{\!\![C]\!\!\}_{\{\leftarrow \text{Goal}\}}$) $\simeq C$,
- ci(C, CI') $\in$ CPs in case lca($CI, \{\!\![C]\!\!\}_{\{\leftarrow \text{Goal}\}}$) $= CI' \not\simeq C$.

The definition of the predicate predicate uses an indeterministic select/3 predicate, i.e.,

```
select(A, [A|B], B).
select(A, [B|C], [B|D]) ← select(A, C, D).
```

a copy_term/2 predicate where copy_term(T_1, T_2) is true if $T_1 \simeq T_2$ and T_1 and T_2 have no variables in common, the anti_unify/3 predicate, and a subsumes_chk/2 predicate where subsumes_chk(T_1, T_2) is true if $T_1 \sqsupseteq T_2$.

Example 14 Consider the following test inputs to the append/3 program.

```
← append([], [], []).
← append([1,2], [], [1,2]).
```

Then we have

```
update_cover([ci(append([], A, A), []),
  ci((append([B|C], D, [B|E])
    ← append(C, D, E)),[])],
  append([], [], []),
  [ci(append([], A, A), append([], [], [])),
  ci((append([B|C], D, [B|E])
    ← append(C, D, E)), [])])

update_cover([ci((append([A|B], C, [A|D])
    ← append(B, C, D)),
  (append([2,3], [4], [2,3,4])
    ← append([3], [4], [3,4]))),
  c(append([], E, E))],
  append([1,2], [], [1,2]),
  [ci((append([A|B], C, [A|D])
    ← append(B, C, D)),
  (append([F,G], H, [F,G|H])
    ← append([G], H, [G|H]))),
  c(append([], E, E))])
```

Program 2 (Cover initialization)

```
get_clauses(Pred, Cs) ←
  findall((Pred ← Body),
          clause(Pred, Body), Cs).

init_instances([], []).
init_instances([C|Cs], [ci(C, []|Ps]) ←
  init_instances(Cs, Ps).
```

Coverage computation will be performed predicate by predicate, so the auxiliary predicates for constructing the appropriate data for the update_cover/3 predicate are as depicted in Program 2.

The get_clauses/2 predicate defines a relation on the set of terms and the set of lists of clauses. It uses the standard findall/3 predicate [4], which, in turn, uses the clause/2 predicate. The latter is a standard built-in predicate in almost all Prolog systems, where clause(Head, Body) is true if Head ← Body is a clause of the the current program.

The init_instances/2 predicate simply initializes the coverage list. The meaning is evident from the clauses.

There is another predicate defining the relation of test coverage in terms of the clauses (to be covered), the set of test inputs, and the coverage. The predicate described here is a bit more extensive as it already provides an interface for a test input generator.

The predicate det_cover/4, as shown in Program 3 defines a relation on

- the set of *partial coverages*, i.e., the set of lists CPs of *coverage*,
- the set of lists of test inputs, T, Gs,
- the set of cover structures, PC.

A *cover structure* is either a constant cover or a term no_cover(*coverage*), indicating that the clause under inspection is covered or the instance *coverage* is covered. The predicate definition for det_cover/4 uses a predicate next_test/4, which serves as an interface for different kinds of providing test inputs. In general, the arguments to next_test/4 are

1. a list of test inputs,
2. a partial coverage, i.e., a list of *coverage*,
3. a test input (intended as the output of next_test/4),

Program 3 (Partial coverage computation)

```
det_cover(CPs, T, Gs, PC) ←
  ( next_test(T, CPs, G, T1) →
    update_cover(CPs, G, CPs1),
    Gs = [G|Gs1],
    ( member(ci(_, _), CPs1) →
      det_cover(CPs1, T1, Gs1, PC)
    ; Gs1 = [],
      PC = cover
    )
  ; Gs = [],
    PC = no_cover(CPs)
  ).
```

4. the tail of the list under 1. (intended to be passed to the next call of next_test/4.

Depending on the definition of next_test/4, test inputs can either be generated or the coverage of a given test inputs set can be calculated. The computation of test inputs can be done using a fair interpreter (cf. [9, 13]) by cutting of the bodies of the clauses to be investigated and solve these goals by the fair interpreter. For coverage determination of a given test input set, next_test/4 can be defined as

```
next_test([G|Gs], _, G, Gs).
```

Here, the predicate essentially defines a relation on lists of test inputs and a goal. next_test/4 selects the first element of a list (of goals), providing it as the next test input, and passes the tail of the list as its last argument. The second argument is not used here, it holds the partial coverage, which can be used in test input generation to determine the next goal.

In the definition of det_cover/4, also the predicate member/2 is used. This predicate defines a relation on the set of terms and the set of lists. member(X, L) is true if X is an element of the list L, i.e.,

```
member(X, [X|Tail]).
member(X, [Y|Tail]) ← member(X, Tail).
```

Summarizing the above predicates,

det_cover(CPs, T, Gs, PC)

is true if

$$\mathrm{PC} = \begin{cases} \text{cover} & : (\forall \mathrm{ci(C, CI)} \in \mathrm{CPs}) \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T \cup CI) \simeq \mathrm{C}, \\ \text{cover} & : \neg(\exists \mathrm{ci(C, CI)} \in \mathrm{CPs}), \\ \text{no\_cover}(NC) & : \text{otherwise}, \end{cases}$$

where $T = \{ \leftarrow G : G \in \mathrm{T} \}$, i.e., we view the list T as a set of goals,

$$\mathrm{Gs} = \mathrm{T},$$

$$CI = \begin{cases} \emptyset & : \mathrm{CI} = [], \\ \{\mathrm{CI}\} & : \text{otherwise}, \end{cases}$$

and NC is a list of elements x such that

$$x = \begin{cases} \mathrm{ci(C, lca}(\{\!\{\mathrm{C}\}\!\}_T \cup CI) & : \mathrm{ci(C, CI)} \in \mathrm{CPs} \text{ and} \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T \cup CI) \not\simeq \mathrm{C}, \\ \mathrm{c(C)} & : \mathrm{ci(C, CI)} \in \mathrm{CPs} \text{ and} \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T \cup CI) \simeq \mathrm{C}, \\ \mathrm{c(C)} & : \mathrm{c(C)} \in \mathrm{CPs}, \end{cases}$$

A special case appears, if

$$\mathrm{CPs} = [\mathrm{ci}(C_1, []), \ldots, \mathrm{ci}(C_m, [])].$$

Then det_cover(CPs, T, Gs, PC) is true if

$$\mathrm{PC} = \begin{cases} \text{cover} & : (\forall \mathrm{ci(C, [])} \in \mathrm{CPs}) \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T) \simeq \mathrm{C}, \\ \text{no\_cover}(NC) & : \text{otherwise}, \end{cases}$$

where NC is a list of elements x such that

$$x = \begin{cases} \mathrm{ci(C, lca}(\{\!\{\mathrm{C}\}\!\}_T) & : \mathrm{ci(C, [])} \in \mathrm{CPs} \text{ and} \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T) \not\simeq \mathrm{C}, \\ \mathrm{c(C)} & : \mathrm{ci(C, [])} \in \mathrm{CPs} \text{ and} \\ & \quad \mathrm{lca}(\{\!\{\mathrm{C}\}\!\}_T) \simeq \mathrm{C}, \end{cases}$$

and, of course, Gs = T. Hence, PC, more precisely the second arguments of its ci/2 terms, represents the coverage of a program P (the first arguments of the ci/2 terms) and the tests inputs of the list T.

The only remaining predicate to complete coverage computation is one defining the det_cover/4 relation for the special case described above. The definition is now obvious and given in Program 4.

Program 4 (Coverage computation)

```
find_cover(Pred, T, Goals, PC) ←
  get_clauses(Pred, Cs),
  init_instances(Cs, CPs),
  det_cover(CPs, T, Goals, PC).
```

6 Discussion and Conclusions

We have developed a test coverage notion for a declarative and relational programming paradigm. This has been

worked out for the logic programming paradigm. The test coverage notion has been based on an abstract model of the program to be tested. We use the set of instances of program clauses induced by program execution and define test coverage by the least upper bound of the clause instances. This is the natural abstract model of a logic program with respect to coverage. Since it uses the space of concretizations of the program built up by concrete execution.

We use anti-unification for determining test coverage. This is the dual to unification which is central for the computational model of logic programming. While unification extracts concrete information out of a program, our test coverage notion uses this concrete information (induced by test inputs) to extract general information (test coverage) via anti-unification.

We showed that the proposed coverage notion possesses all the reasonable properties which are required from a coverage notion, notably that every program has a cover, that the coverage measure is monotone, and exhaustive testing always covers the program under inspection. To our knowledge, no formal notion of test coverage for logic programs has been proposed yet. Thus, a comparison with other work on this topic could not be done.

Acknowledgement

This work has been supported by Mitsubishi Electric Europe, project *Modelling and Implementation of a Flexible Robot Cell.*

References

[1] A. Azem, F. Belli, and O. Jack. Implementation-oriented testing and reliability determination of Prolog programs. In *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, pages 526–533, San Francisco Bay, June 1993.

[2] F. Belli and O. Jack. Implementation-based analysis and testing of Prolog programs. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 70–80. ACM Press, 1993.

[3] F. Belli and O. Jack. A product assurance environment for Prolog. In Ducassé et al. [5], pages 94–99.

[4] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin Heidelberg, 1987.

[5] M. Ducassé, B. Le Charlier, Y.-J. Lin, and U. Yalcinalp, editors. *Proceedings of ILPS'93 Workshop on Logic Programming Environments, LPE'93*, Vancouver, October 1993. Publication IRSIA, Campus de Beaulieu, 35042 Rennes, France.

[6] J. Gannon. Testing tools using formal specification and coverage metrics. *GI Softwaretechnik-Trends*, 6(1):5–11, 1986.

[7] G. Huet. *Resolution d'equations dans des languages d'ordre* $1, 2, \ldots, \omega$. These d'etat, Universite de Paris VII, 1976.

[8] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kauffman, 1988.

[9] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin Heidelberg, 1987.

[10] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1(2):303–338, 1990.

[11] E. F. Miller. Automated generation of testcase data sets. In *Proceedings of the International Conference on Reliable Software*, pages 52–58, 1975.

[12] E. F. Miller. Testing and test documentation (workshop report). *IEEE Computer*, March 1979.

[13] R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, April 1985.