Automatic Generation of Test Inputs for Mercury

François Degrave[1], Tom Schrijvers[2], and Wim Vanhoof[1]

[1] Faculty of Computer Science, University of Namur, Belgium
{fde,wva}@info.fundp.ac.be
[2] Department of Computer Science, K.U. Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. In this work, we consider the automatic generation of test inputs for Mercury programs. We use an abstract representation of a program that allows to reason about program executions as paths in a control-flow graph. Next, we define how such a path corresponds to a set of constraints whose solution defines input values for the predicate under test such that when the predicate is called with respect to these input values, the execution is guaranteed to follow the given path. The approach is similar to existing work for imperative languages, but has been considerably adapted to deal with the specificities of Mercury, such as symbolic data representation, predicate failure and non-determinism.

1 Introduction

It is a well-known fact that a substantial part of a software development budget (estimates range from 50% to 75% [1]) is spent in *corrective maintenance* – the act of correcting errors in the software under development. Arguably the most commonly applied strategy for finding errors and thus producing (more) reliable software is testing. Testing refers to the activity of running a software component with respect to a well-chosen set of inputs and comparing the outputs that are produced with the expected results in order to find errors. While the fact that the system under test passes successfully a large number of tests does not prove correctness of the software, it nevertheless increases confidence in its correctness and reliability [2].

In testing terminology, a *test case* for a software component refers to the combination of a single test input and the expected result whereas a *test suite* refers to a collection of individual test cases. Running a test suite is a process that can easily be automated by running the software component under test once for each test input and comparing the obtained result with the expected result as recorded in the testcase.

The hard part of the testing process is *constructing* a test suite, which comprises finding a suitable set of test inputs either based on the specification (*black-box* testing) or on the source code of program (*whitebox* or *structural* testing). In the latter approach, which we follow in this work, the objective is to create a set of test inputs that cover as much source code as possible according to some *coverage criterion*; some well-known examples being statement, branch and path coverage [3].

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 71–86, 2009.

Although some work exists on automatic generation of test inputs in functional and logic programming languages [4,5,6,7], most attempts to automate the generation of test inputs have concentrated on imperative [8,9,10] or object oriented programs [11,12]. In this work we consider the automatic generation of test inputs for programs written in the logic programming language Mercury [13]. Although being a logic programming language, Mercury is strongly moded and allows thus to generate a complete control flow graph similar to that of imperative programs which in turn makes it possible to adapt techniques for test input generation originally developed in an imperative setting. Such a control-flow based approach is appealing for the following reasons: (1) Given a path in the graph that represents a possible execution for a predicate, one can compute input values such that when the predicate is called with respect to those values, its execution will follow the derivation represented by the given path. These input values can easily be converted in a *test case* by the programmer: it suffices to add the expected output for the predicate under consideration. (2) Given a (set of) program point(s) within the graph, it is possible to compute a path through the graph that ressembles a computation covering that (set of) program point(s). By extension, this allows for algorithms to compute a set of execution paths (and thus test cases) guided by some coverage criterion [3].

The approach we present is similar in spirit to the constraint based approach of [8], although considerably adapted and extended to fit the particularities of a logic programming language. In particular, the fact that our notion of execution path captures failures and the sequence of answers returned by a nondeterministic predicate allows to automatically generate test inputs that test the implementation strategy of possibly failing and non-deterministic predicates. In addition, since *all* dataflow in a Mercury program can easily be represented by constraints on the involved variables implies that our technique is, in contrast to [8], not limited to numerical values. The current work is a considerably extended and revised version of the abstract presented at LOPSTR'06 [14].

2 Preliminaries

Mercury is a statically typed logic programming language [13]. Its type system is based on polymorphic many-sorted logic and essentially equivalent to the Mycroft-O'Keefe type system [15]. A type definition defines a possibly polymorphic type by giving the set of function symbols to which variables of that type may be bound as well as the type of the arguments of those functors [13]. Take for example the definition of the well known polymorphic type $list(T)$:

```
:- type list(T) ---> [] ; [T|list(T)].
```

According to this definition, if T is a type representing a given set of terms, values of type `list(T)` are either the empty list `[]` or a term $[t_1|t_2]$ where t_1 is of type T and t_2 of type $list(T)$.

In addition to these so-called *algebraic types*, Mercury defines a number of primitive types that are builtin in the system. Among these are the *numeric types* `int` (integers) and `float` (floating point numbers). Mercury programs are

statically typed: the programmer declares the type of every argument of every predicate and from this information the compiler infers the type of every local variable and verifies that the program is well-typed.

In addition, the Mercury *mode system* describes how the instantiation of a variable changes over the execution of a goal. Each predicate argument is classified as either input (ground term before and after a call) or output (free variable at the time of the call that will be instantiated to a ground term). A predicate may have more than one mode, each mode representing a particular usage of the predicate. Each such mode is called a *procedure* in Mercury terminology. Each procedure has a declared (or inferred) *determinism* stating the number of solutions it can generate and whether it can fail. Determinisms supported by Mercury include `det` (a call to the procedure will succeed exactly once), `semidet` (a call will either succeed once or fail), `multi` (a call will generate one or more solutions), and `nondet` (a call can either fail or generate one or more solutions)[1]. Let us consider for example the definition of the well-known `append/3` and `member/2` predicates. We provide two mode declarations for each predicate, reflecting their most common usages:

```
:- pred append (list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
append([], Y, Y).
append([E|Es], Y ,[E|Zs]):- append(Es, Y, Zs).

:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.
member(X, [X|_]).
member(X, [Y|T]) :- not (X=Y), member(X, T).
```

For `append/3`, either the first two arguments are input and the third one is output in which case the call is deterministic (it will succeed exactly once), or the third argument is input and the first two are output in which case the call may generate multiple solutions. Note that no call to `append/3` in either of these modes can fail. For `member/2`, either both arguments are input and the call will either succeed once or fail, or only the second argument is input, in which case the call can fail, or generate one or more solutions.

3 Extracting Execution Paths

3.1 A Control Flow Graph for Mercury

For convenience, we consider that a Mercury program consists of a set of distinct procedures (a multi-moded predicate should have been transformed into different procedures). Each procedure should be well-typed and well-moded, and be in *superhomogeneous form*. A procedure in superhomogeneous form consists

[1] There exist other modes and determinisms but they are outside the scope of this paper; we refer to [13] for details

of a single clause (usually a disjunction) in which the arguments in the head of the clause and in procedure calls in the body are all distinct variables. Explicit unifications are generated for these variables in the body, and complex unifications are broken down into simple ones. Moreover, using mode information each unification is classified as either a test between two atomic values X==Y, an assignement Z:=X, deconstruction $X \Rightarrow f(Y_1, ..., Y_n)$ or construction $X \Leftarrow f(Y_1, ..., Y_n)$. See [13] for further details. We associate a distinct *label* to a number of program points of interest. These labels – which are written in subscripts and attached to the left and/or the right of a goal – are intended to identify the nodes of the program's control flow graph.

Definition 1. *Let Π denote the set of procedures symbols, Σ the set of function symbols and $\mathcal{V}$ and $\mathcal{L}$ respectively the set of variables and labels in a given program P. The syntax of a program in labelled superhomogenous form is defined as follows:*

$$
\begin{array}{lll}
LProc & ::= & p(X_1, \ldots, X_k) \ \texttt{:-}\ LConj. \\
LConj\ C & ::= & {}_lG_{l'} \mid {}_lG\texttt{,}C \\
LDisj\ D & ::= & C\texttt{;}C'' \mid D\texttt{;}C \\
LGoal\ G & ::= & A \mid D \mid \texttt{not}(C) \mid \texttt{if}\ C\ \texttt{then}\ C'\ \texttt{else}\ C'' \\
Atom\ A & ::= & X\texttt{==}Y \mid X \Rightarrow f(Y_1, ..., Y_n) \mid X \Leftarrow f(Y_1, ..., Y_n) \\
 & & \mid\ Z\texttt{:=}X \mid p(X_1, ..., X_n)
\end{array}
$$

where X, Y, Z and $X_i, Y_i (0 \leq i \leq n) \in \mathcal{V}, p/k \in \Pi, f \in \Sigma, l, l' \in \mathcal{L}$. All labels within a given program are assumed to be distinct.

Note that according to the definition above, a label is placed between two successive conjuncts, as well as at the beginning and at the end of a conjunction and a disjunction.

Example 1. The `append(in,in,out)`, `member(in,in)` and `member(out,in)` procedures in labelled superhomogeneous form look as follows. Note that the only difference between the two procedures for `member` is the use of test, respectively an assignment at program point l_3.

$$
\begin{array}{l}
\texttt{append(X :: in, Y :: in, Z :: out)} : - \\
\quad {}_{l_1}({}_{l_2}\texttt{X} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_3}\ \texttt{append(E}_\texttt{s}\texttt{, Y, W)}_{,l_4}\ \texttt{Z} \Leftarrow [\texttt{E}|\texttt{W}]_{l_5}\ \texttt{;}\ {}_{l_6}\texttt{Z} = \texttt{Y}_{l_7})_{l_8}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{member(X :: in, Y :: in)} : - \\
\quad {}_{l_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_2}\ ({}_{l_3}\texttt{X} == \texttt{E}_{l_4}\ \texttt{;}\ {}_{l_5}\texttt{member(X, E}_\texttt{s})_{l_6})_{l_7}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{member(X :: out, Y :: in)} : - \\
\quad {}_{l_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_2}\ ({}_{l_3}\texttt{X} := \texttt{E}_{l_4}\ \texttt{;}\ {}_{l_5}\texttt{member(X, E}_\texttt{s})_{l_6})_{l_7}.
\end{array}
$$

In [16], we have defined how one can build and use a control flow graph for Mercury. Given a Mercury program in labelled superhomogeneous form, its control graph can easily be constructed as follows. The nodes of the directed graph are the labels occuring in the program, together with two special labels: a *success label* l_S and a *failure label* l_F, representing respectively success and failure of a (partial) derivation. The graph contains three types of arcs:

- Two nodes l and l' are linked with a *regular arc* if one of the following conditions holds:
  1. l is the label preceding and l' the label succeeding an atom;
  2. l is the label preceding an atom that can possibly fail (i.e. deconstruction or equality test) and which is not part of the condition of a *if-then-else* construction and l' is the failure label l_F;
  3. l is the label preceding an atom that can possibly fail in the condition of a *if-then-else* construction and l' is the first label of the goal in the *else* part of this construction;
  4. (a) l is the label preceding a disjunction and l' is the label preceding one of the disjuncts inside this disjunction; or
     (b) l is the label succeeding one of the disjuncts inside a disjunction and l' is the label succeeding the disjunction as a whole;
  5. l is the label preceding a procedure call and l' is the first label of this procedure's body goal;
  6. l is the last label of the labelled program and l' is the success label l_S.
- Two nodes l and l' are linked with a *return-after-success* or *return-after-failure* arc, denoted $(l, l')^{rs}$ respectively $(l, l')^{rf}$, if l precedes a procedure call and if the execution should be resumed at l' upon success, respectively failure, of the call.

Moreover, regular arcs are annotated by a natural number called *priority*. Each arc initiating a disjunct is annotated by the *position* of the disjunct in the disjunction when counted from right to left. Other arcs are annotated by zero.

Example 2. Figure 1. depicts two control flow graphs. The left one corresponds to a program defining the `member(in,in)` procedure, the right one defining the `member(out,in)` procedure, as defined in Example 1.

In both graphs, the arc (l_1, l_2) represents success of the atom $Y \Rightarrow [E|E_s]$ whereas the arc (l_1, l_F) represents failure of the atom. In the first case, the execution continues at l_2, in the latter it fails. The only difference between both graphs is the presence of the arc (l_3, l_F) in the graph for `member(in,in)`; it represents the fact that the atom at l_3 (the test X==E) can fail whereas the assignement X:=E in `member(out,in)` cannot. In order to avoid overloading the figures, we

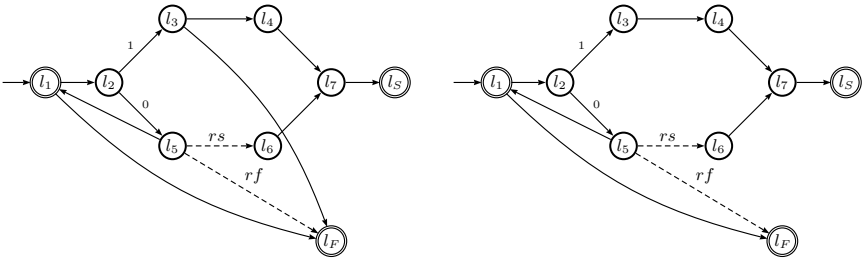


Fig. 1. `member(in,in)` and `member(out,in)`

depict priorities only when relevant, i.e. when they annotate an arc representing the entry into a disjunct. As such, a node from which leave several arcs bearing different priorities represents effectively a *choicepoint*. When walking through the graph in order to collect execution paths, the highest priority must be chosen first. This denotes the fact that, operationally, in a disjunction $(D_1; D_2)$ the disjunct D_1 is executed first, and D_2 is executed only if a backtracking occurs.

3.2 Deriving Execution Sequences

A program's control flow graph allows to reason about *all* possible executions of a given procedure. We first define the notion of a *complete execution segment* that represents a straightforward derivation from a call to a success (and thus the production of an answer) or a failure, in which an arbitrary disjunct is chosen at each encountered choicepoint. The definition is in two parts:

Definition 2. *An* execution segment *for a procedure p is a sequence of labels with the first label being the label appearing at the very beginning of p's body, the last label either the success label l_S or the failure label l_F, where for each pair of consecutive labels (l_i, l_{i+1}) the following conditions hold:*

1. *If $l_i \neq l_S$ and $l_i \neq l_F$ then l_i is connected to l_{i+1} in the graph with a regular arc.*
2. *If $l_i = l_S$ then there exists l_c $(c < i)$ such that l_c and l_{i+1} are connected in the graph with a* return-after-success *arc, and the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment;*
3. *If $l_i = l_F$ then there exists l_c $(c < i)$ such that l_c and l_{i+1} are connected in the graph with a* return-after-success *arc, the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment* and *each pair of consecutive labels (l_j, l_{j+1}) with $c + 1 \leq j \leq i$ is connected in the graph with a regular arc of which priority equals zero;*

Definition 2 basically states that an execution segment is a path trough the graph in which a label l_{i+1} follows a label l_i if both labels are connected by a regular arc (condition (1)). If, however, l_i represents the exit from a procedure call – either by success (condition (2)) or failure (condition (3)) – then the next label should be a valid resume point. Moreover, conditions 2 and 3 impose that each return has a corresponding call, and guarantee that the sequence of labels representing the execution through the callee is a valid execution segment as well. Condition 3 also denotes that the return after the *failure* of a call can be performed only if the corresponding call definitely failed, i.e. it is impossible to perform backtracking to a choicepoint created after the call that would make the latter succeed. In order to be useful, an execution segment must be *complete*, intuitively meaning that there should be no calls without a corresponding return, unless the derivation ends in failure and contains unexplored alternatives for backtracking.

Definition 3. *An execution segment S for a procedure p is* complete *if the following conditions hold:*

1. *If S ends with l_S, then no suffix of S is an execution segment for any procedure of the program;*
2. *If S ends with l_F then if S has a suffix S' which is an execution segment for a procedure of the program, then S' contains at least one pair of consecutive labels (l_j, l_{j+1}) connected in the graph by a regular arc annotated by $n \geq 1$.*

In the above definition, condition 1 guarantees that, in a derivation leading to a success, every call has a corresponding return, while condition 2 imposes that, in a derivation leading to a failure, if there exists a call with no corresponding return, it must be possible to backtrack inside this call. Note that Definitions 2 and 3 only allow for *finite* (complete) execution segments.

Example 3. Let us consider `member(in,in)`, defined in Example 1. The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the left in Figure 1. It corresponds to the execution of a call in which the deconstruction of the list succeeds, the first disjunct is chosen at the choicepoint l_2, and the equality test between the first element and the call's first argument also succeeds, leading to the success of the predicate. In other words, it represents a call `member(X,Y)` in which the element X appears at the first position in the list Y.

Example 4. Let us now consider the nondeterministic `member(out,in)` procedure, also defined in Example 1. The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the right in Figure 1. The execution segment s represents the execution leading to the first solution of a call `member(X,Y)` in which the list Y is not empty.

Of particular interest are the choices commited at each choicepoint encountered along a given complete execution segment. In the remaining we represent these choices by a sequence of integers, which are the priorities of the arcs chosen at each choicepoint.

Definition 4. *Let $s = \langle l_1, ..., l_n \rangle$ be a complete execution segment. The* sequence of choices *associated to s, noted $\mathcal{SC}(s)$, is defined as follows:*
$\mathcal{SC}(\langle l_1 \rangle) = \langle\rangle$
$\mathcal{SC}(\langle l_1, ..., l_n \rangle) = \mathcal{P}rior(l_1, l_2) \cdot \mathcal{SC}(\langle l_2, ..., l_n \rangle)$
where $\cdot$ denotes sequence concatenation and $\mathcal{P}rior(l_i, l_{i+1}) = \langle nb \rangle$ if l_i is a choicepoint and l_i and l_{i+1} are connected in the graph with a regular arc annotated by a number nb, or $\langle\rangle$ if l_i is not a choicepoint.

Example 5. Let us consider the complete execution segment $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ for `member(in,in)`, defined in Example 3. The sequence of choices associated to this segment is $\mathcal{SC}(s) = \langle 1 \rangle$. On the other hand, for the complete execution segment $s' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$, representing an execution in which the first argument of the call to member occurs in the second position of its second argument, we have $\mathcal{SC}(s') = \langle 0, 1 \rangle$.

A complete execution segment for a procedure p represents a single derivation for a call to p with respect to some (unknown) input values in which for each

encountered choicepoint an arbitrary choice is made. In order to model a real execution of the procedure, several such derivations need in general to be combined, in the right order. The order between two complete execution segments is determined by the sequence of choices that have been made. The sequence of choices being a sequence over natural numbers, we define the following operation:

Definition 5. *Let* $\langle i_1, ..., i_m \rangle$ *denote a sequence over* $\mathbb{N}$*, we define*

$$decr(\langle i_1, ..., i_m \rangle) = \begin{cases} \langle i_1, ..., (i_m - 1) \rangle & \text{if } i_m > 0 \\ decr(\langle i_1, ..., i_{m-1} \rangle) & \text{otherwise} \end{cases}$$

For a sequence of choices $s = \langle i_1, \ldots, i_n \rangle$, $decr(s)$ represents a new sequence of choices that is obtained from s by deleting the rightmost zeros and decrementing the rightmost non-zero choice by one. Operationally, $decr(s)$ represents the stack after performing a backtrack operation.

Definition 6. *An* execution sequence *for a procedure p is defined as a sequence of complete execution segments* $\langle S_1, ..., S_n \rangle$ *for p having the following properties:*

1. *For all* $0 < i < n$, $decr(\mathcal{SC}(S_i))$ *is a prefix of* $\mathcal{SC}(S_{i+1})$*;*
2. *It is not possible to derive from the graph a complete execution segment* S_k *for the procedure such that* $decr(\mathcal{SC}(S_k))$ *is a prefix of* $\mathcal{SC}(S_1)$*.*

Note that an execution sequence $\langle S_1, \ldots, S_n \rangle$ represents a derivation *tree* for a call to the predicate under consideration with respect to some (unknown) input values. Indeed, the first segment S_1 represents the first branch, i.e. the derivation in which for each encountered choicepoint the first alternative is chosen (the one having the highest priority in the graph). Likewise, an intermediate segment S_{i+1} $(i \geq 1)$, represents the same derivation as S_i except that at the last choicepoint having an unexplored alternative, the next alternative is chosen. Note that the derivation tree represented by $\langle S_1, \ldots, S_n \rangle$ is not necessarily complete. Indeed, the last segment S_n might contain choicepoints having unexplored alternatives. However, by construction, there doesn't exist a complete execution segment representing an unexplored alternative between two consecutive segments S_i and S_{i+1}.

While the definition allows in principle to consider infinite execution sequences, an execution sequence cannot contain an infinite segment, nor can it contain a segment representing a derivation in which one of the choicepoints has a previous alternative that would have led to an infinite derivation. It follows that an execution sequence represents a finite part of a real execution of the Mercury procedure under consideration (always with respect to the particular but unknown input values). The attentive reader will notice that if $\mathcal{SC}(S_n)$ is a sequence composed of all zeros, then the execution sequence represents a complete execution in which all answers for the call have been computed.

Example 6. Reconsider the nondeterministic `member(out,in)` procedure and the following complete execution segments:

$$\begin{aligned} S_1 &= \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle, \\ S_2 &= \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle, \\ S_3 &= \langle l_1, l_2, l_5, l_1, l_2, l_5, l_1, l_F, l_F, l_F \rangle \rangle \end{aligned}$$

The reader can easily verify that $\mathcal{SC}(S_1) = \langle 1 \rangle$, $\mathcal{SC}(S_2) = \langle 0, 1 \rangle$, and $\mathcal{SC}(S_3) = \langle 0, 0, 1 \rangle$. Obviously, $decr(\mathcal{SC}(S_1)) = \langle 0 \rangle$ is a prefix of $\mathcal{SC}(S_2)$ and $decr(\mathcal{SC}(S_2)) = \langle 0, 0 \rangle$ is a prefix of $\mathcal{SC}(S_3)$. Moreover, there does not exist a complete execution segment s such that $decr(s)$ is a prefix of S_1 and hence $\langle S_1, S_2, S_3 \rangle$ is an execution sequence for `member(out,in)`.

The execution sequence from Example 6 corresponds to the execution of a call $member(X, Y)$ in which a first solution is produced by assigning the first element of the list Y to X and returning from the call (expressed by the first segment of the path, ending in l_S). A second solution is produced by backtracking, choosing the disjunct corresponding to l_5, performing a recursive call, assigning the second element of the list to X, and performing the return (the second segment, also ending in l_S). The execution continues by backtracking and continuing at l_5 and performing a recursive call in which the deconstruction of the list argument fails. In other words, the execution sequence e corresponds to a call to member in which the second argument is instantiated to a list containing exactly two elements.

In te remaining, we show how to compute input values from an execution sequence. Our approach consists of two phases. First, an execution sequence is translated into a set of constraints on the procedure's input (and output) arguments. Next, a solver written in CHR is used to generate arbitrary input values that satisfy the set of constraints. The solver contains type information from the program under test, but can be automatically generated from the program.

4 From Execution Sequences to Sets of Constraints

Since Mercury programs deal with both symbolic and numeric data, we consider two types of constraints: *symbolic constraints* which are either of the form $x = f(y_1, \ldots, y_n)$ or $x \not\Rightarrow f$[2] and *numerical constraints* which are of the form $x = y \oplus z$ (with $\oplus$ an arithmetic operator). Furthermore we consider constraints of the form $x = y$ and $x \neq y$ that can be either symbolic or numeric. Note that as a notational convenience, constraint variables are written in lowercase in order to distinguish them from the corresponding program variables.

In the remaining we assume that, in the control flow graph, edges originating from a label associated to an atom are annotated as follows: in case of a predicate call the edge is annotated by the call itself; in case of a unification it is annotated by the corresponding constraint, depending of the kind of atom and whether it succeeds or fails, as follows:

source program	(l, l')	(l, l'') with $l' \neq l''$
${}_l X := Y_{l'}$	$x = y$	not applicable
${}_l X == Y_{l'}$	$x = y$	$x \neq y$
${}_l X <= f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	not applicable
${}_l X => f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	$x \not\Rightarrow f$
${}_l X := Y \oplus Z_{l'}$	$x = y \oplus z$	not applicable

[2] The constraint $x \not\Rightarrow f$ denotes that the variable x cannot be deconstructed into a term of which the functor is f. Formally, that means $\forall \overline{y} : \; x \neq f(\overline{y})$.

In order to collect the constraints associated to an execution segment, the basic idea is to walk the segment and collect the constraints associated to the corresponding edges. However, the constraints associated to each (sub)sequence of labels corresponding to the body of a call need to be appropriately renamed. Therefore, we keep a *sequence* of renamings during the constraint collection phase, initially containing a single renaming (possibly the identity renaming). Upon encountering an edge corresponding to a predicate call, a fresh variable renaming is constructed and added to the sequence. It is removed when the corresponding return edge is encountered. As such, this sequence of renamings can be seen as representing the call stack, containing one renaming for each call in a chain of (recursive) calls.

Definition 7. *Let E denote the set of edges in a control flow graph and let $\langle l_1, \ldots, l_n \rangle$ be an execution segment for a procedure p. Given a sequence of renamings $\langle \sigma_1, \ldots, \sigma_k \rangle$, we define $\mathcal{U}(\langle l_1, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$ as the set of constraints C defined as follows :*

1. *if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \notin E$ then let c be the constraint associated to the edge (l_1, l_2). We define $C = \{\sigma_1(c)\} \cup \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$*
2. *if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \in E$ then let $p(X_1, \ldots, X_m)$ be the call associated to the edge (l_1, l_2). If $head(p) = p(F_1, \ldots, F_m)$ then let γ be a new renaming mapping f_i to x_i (for $1 \leq i \leq m$), and mapping every variable occuring free in $body(p)$ to a fresh variable. Then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \gamma, \sigma_1, \ldots, \sigma_k \rangle)$.*
3. *if $l_1 = l_S$ or $l_1 = l_F$, then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_2, \ldots, \sigma_k \rangle)$.*

Furthermore, we define $\mathcal{U}(\langle \rangle, \langle \rangle) = \emptyset$.

Note that the three cases in the definition above are mutually exclusive. The first case treats a success or failure edge associated to a unification. It collects the corresponding constraint, renamed using the *current* renaming (which is the first one in the sequence). The second case treats a success arc corresponding to a predicate call, by creating a fresh renaming γ and collecting the constraints on the remaining part of the segment after adding γ to the sequence of renamings. The third case, representing a return from a call, collects the remaining constraints after removing the current renaming from the sequence of renamings such that the remaining constraints are collected using the same renamings as those before the corresponding call.

Example 7. Let us reconsider the procedure `member(in,in)` and the execution segment $s' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$ given in Example 5. If we assume that *id* represents the identity renaming and that, when handling the recursive call at l_5, the constraint variables e and es, corresponding to the local variables of `member`, are renamed into e' and es', we have

$$\mathcal{U}(s', \langle id \rangle) = \{y = [e|es], x \neq e, es = [e'|es'], x = e'\}.$$

As can be seen from Example 7, the set of constraints associated to an execution segment s defines the *minimal* instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by s. In case of Example 7 we have $y = [e, x|es] \wedge x \neq e$. Indeed, whatever (type correct) further instantiation we choose for the variables x, e and es, as long as the above condition is satisfied, the execution of $member(x, y)$ is guaranteed to follow the execution segment s. A test input can thus be computed for a given execution segment by solving the associated set of constraints, further instantiating the free variables by arbitrary values, as long as the instantiation remains type correct.

To collect the constraints associated to an execution sequence, it suffices to collect the constraints associated to each individual execution segment using an appropriate initial renaming in order to avoid nameclashes.

Definition 8. *Let $\overline{S} = \langle s_1, \ldots, s_n \rangle$ denote an execution sequence for a procedure p. The* set of constraints *associated to $\overline{S}$, denoted $\mathcal{C}(\overline{S})$, is defined as*

$$\mathcal{C}(\langle s_1, \ldots, s_n \rangle) = \bigcup_{1 \leq i \leq n} \mathcal{U}(s_i, \sigma_i)$$

where each σ_i is a renaming mapping each non-input variable of p to a fresh variable name.

The initial renamings do not change the name of the procedure's input variables. Indeed, since each segment represents a different derivation for the *same* input values, *all* constraints on these values from the different segments must be satisfied.

Example 8. Let $\overline{S} = \langle S_1, S_2, S_3 \rangle$ be the execution sequence defined in Example 6 for the `member(out,in)` procedure defined in Section 2. Assuming that an initial renaming σ_i simply adds the index i to all concerned variables, and assuming that when handling the recursive call variables e and es are renamed into e' and es', one can easily verify that the set of constraints associated to $\overline{S}$ is as follows:

$$\begin{aligned}\mathcal{C}(\langle S_1, S_2, S_3 \rangle) &= \mathcal{U}(S_1, \sigma_1) \cup \mathcal{U}(S_2, \sigma 2) \cup \mathcal{U}(S_3, \sigma_3) \\ &= \{y = [e_1|es_1], x_1 = e_1\} \\ &\quad \cup \{y = [e_2|es_2], es_2 = [e'_2|es'_2], x_2 = e'_2\} \\ &\quad \cup \{y = [e_3|es_3], es_3 = [e'_3|es'_3], es'_3 \not\approx [\ |\ ]\}\end{aligned}$$

For a given execution sequence $\overline{S}$, $\mathcal{C}(\overline{S})$ defines the minimal instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by $\mathcal{S}$. In Example 8 above, the set of constraints $\mathcal{C}(\langle S_1, S_2, S_3 \rangle)$ implies

$$y = [e_1, e'_2|es'_3] \wedge es'_3 \not\approx [\ |\ ] \wedge x_1 = e_1 \wedge x_2 = e'_2$$

and, indeed, whatever type correct instantiation we choose for the variables e_1 and e'_2, we will always have es'_3=[] and the execution of a call $member(\_, [E_1, E'_2])$ is guaranteed to proceed along the specified path.

Note that the obtained constraint set defines, for each segment ending in success, the minimal instantiation of the procedure's *output* arguments as well.

In Example 8, the sequence of output arguments is given by $\langle x_1, x_2 \rangle$. Hence, the computed results could be automatically converted not only into test inputs but into complete test cases. Of course, before such a computed test case can be recorded for further usage, the programmer should verify that the computed output corresponds with the *expected* output.

5 Constraint Solving

The constraints of a path are either *satisfiable* or *unsatisfiable*. The latter means that one or more labels in the path cannot be reached along the path (but may be reached along other paths). The latter means that solutions (one or more) exist, and that they will exercise the execution path. In order to establish the satisfiability, we take the usual constraint programming approach of interleaving *propagation* and *search*.

Propagation We reuse existing (CLP) constraints for most of our base constraints.

- $x = y$ and $x = f(\overline{y})$ are implemented as unification,
- $x \neq y$ is implemented as the standard Herbrand inequality constraint, known as `dif/2` in many Prolog systems, and
- $x = y \oplus z$ is implemented as the corresponding CLP(FD) constraint.

For $x \not\Rightarrow f$ we have our custom constraint propagation rules, implemented in CHR, based on the domain representation of CLP(FD). However, rather than maintaining a set of possible values for a variable, the domain of a variable is the set of possible function symbols. The initial domain are all the function symbols of the variable's type. For instance, the contraint `domain(X,{[]/0,[|]/2})` expresses that the possible functions symbol for variable `X` with type `list(T)` are `[]/0` and `[|]/2`, which is also its initial domain.

The following CHR rules further define the constraint propagators (and simplifiers) for the `domain/2` constraint:

```
domain(X,∅)       ==> fail.
domain(X,{F/A})   <=> functor(X,F,A).
domain(X,D)       <=> nonvar(X) | functor(X,F,A), F/A ∈ D.
domain(X,D1), domain(X,D2) <=> domain(X,D1 ∩ D2).
domain(X,D), X ↛ F/A       <=> domain(X,D \ {F/A}).
```

Search Step During search, we enumerate candidate values for the undetermined terms. From all undetermined terms, we choose one x and create a branch in the search tree for each function symbol f_i in its domain. In branch i, we add the constraint $x = f_i(\overline{y})$, where $\overline{y}$ are fresh undetermined terms. Subsequently, we exhaustively propagate again. Then either an (1) inconsistency is found, (2) all terms are determined or (3) some undetermined terms remain. In case (1) we must explore other branches, and in case (2) we have found a solution. In case (3) we simply repeat with another Search Step.

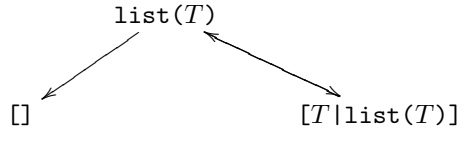
Table 1. Test input generation for `member(X::in,Y::in)` and `member(X::out,Y::in)`

Test inputs		Computed
`X::in`	`Y::in`	Result
0	[0]	Success
0	[1, 0]	Success
0	[1, 1, 0]	Success
1	[0, 0]	Failure
1	[0]	Failure
0	[]	Failure

Test input	Computed Result
`Y::in`	`X::out`
[0, 1, 2]	0, 1, 2
[0, 1]	0, 1
[0]	0
[]	–

Our search algorithm visits the branches in depth-first order. Hence, we must make sure not to get caught in an infinitely diverging branch of the search tree, e.g. one that generates a list of unbounded length. For this purpose, we order a type's functions symbols according to the *type graph*. The nodes of a type graph consist of types and function symbols. There is an edge in the graph from each type to its function symbols, and from each function symbol to its argument types. We order the function symbols of a type according to the order in which they appear in a topologic ordering of the type graph.[3]

Example 9. Consider the list type `:- type list(T) --> [] ; [T|list(T)]`. Its type graph is:



A topological order is $\langle$`[]`, `list(`T`)`, `[`T`|list(`T`)]`$\rangle$, which yields the function symbol ordering $\langle$`[]/0`, `[|]/2`$\rangle$. Indeed, if we first try `[|]/2`, the search will diverge.

To conclude this section, note that our approach is the opposite of type inference, as we infer terms from types rather than types from terms. Among the vast amount of work on type inference we note [17] which employs a similar function symbol domain solver, for resolving ad-hoc overloading.

6 Implementation and Evaluation

The described approach for generating test inputs was implemented in Mercury. Our implementation first constructs a control flow graph for the program under test, and computes a set of execution sequences for the procedures in the program. To keep the set of execution sequences finite, we use a simple termination

[3] We assume that all types are well-founded, e.g. the type `:- type stream(T) --> cons(T,stream(T)).` is not a valid type.

Table 2. Test cases generation for `bubblesort(in,out)`

Test input	Computed Result	Test input	Computed Result	Test input	Computed Result
List::in	Sorted::out	List::in	Sorted::out	List::in	Sorted::out
[]	[]	[2, 1, 0]	[0, 1, 2]	[1, 2, 1, 0]	[0, 1, 1, 2]
[0]	[0]	[0, 0, 0, 0]	[0, 0, 0, 0]	[1, 1, 0, 0]	[0, 0, 1, 1]
[0, 0]	[0, 0]	[0, 0, 1, 0]	[0, 0, 0, 1]	[2, 2, 1, 0]	[0, 1, 2, 2]
[1, 0]	[0, 1]	[0, 1, 1, 0]	[0, 0, 1, 1]	[1, 0, 0, 0]	[0, 0, 0, 1]
[0, 0, 0]	[0, 0, 0]	[1, 1, 1, 0]	[0, 1, 1, 1]	[2, 0, 1, 0]	[0, 0, 1, 2]
[0, 1, 0]	[0, 0, 1]	[1, 0, 1, 0]	[0, 0, 1, 1]	[2, 1, 1, 0]	[0, 1, 1, 2]
[1, 1, 0]	[0, 1, 1]	[0, 1, 0, 0]	[0, 0, 0, 1]	[2, 1, 0, 0]	[0, 0, 1, 2]
[1, 0, 0]	[0, 0, 1]	[0, 2, 1, 0]	[0, 0, 1, 2]	[3, 2, 1, 0]	[0, 1, 2, 3]

Table 3. Test cases generation for different procedures

Procedures	Determinism	Maximum call depth	Solutions requested	Number of test cases	Execution time (in ms)
Partition(in,in,out,out)	det	6	-	126	890
Append(in,in,out)	det	6	-	6	40
Append(out,out,in)	nondet	6	10	10	70
Doubleapp(in,in,in,out)	det	3	-	6	650
Doubleapp(out,out,out,in)	multi	6	8	4	4670
Member(in,in)	semidet	5	-	12	700
Member(out,in)	nondet	5	5	6	1310
Applast(in,in,out)	det	3	-	14	40
Match(in,in)	semidet	3	-	6	960
Matchappend(in,in)	semidet	4	-	20	90
MaxLength(in,out,out)	det	5	-	10	600
Revacctype(in,in,out)	det	4	-	12	500
Transpose(in,out)	det	2	-	9	1370

scheme that consists in limiting the call depth as well as the the number of solutions in each execution sequence (in the case of non-deterministic procedures). Since our implementation is meant to be used as a proof of concept, performance of the tool has not been particularily stressed.

Table 1 gives the test inputs that are generated for the `member(in,in)` and `member(out,in)` procedures defined in Example 1 when the call depth is limited to 2 and the number of solutions in an execution sequence to 3. For `member(in,in)` we indicate for each generated test input whether this test input makes the procedure succeed or fail. For `member(out,in)` we give for each generated test input the corresponding output values.[4] As described in Section 4, it is up to the user to check whether the obtained result corresponds to the expected result when creating the test suite. The test inputs (and corresponding outputs) presented in Table 1 were generated in 20 ms , respectively 10 ms.

Table 2 contains the generated test inputs for a procedure implementing the bubble-sort algorithm. This well-know algorithm for list sorting uses two

[4] In the case of `member(out,in)`, we added manually the constraint `all_different/1` which guarantees all the elements of the list to be different.

recursive sub-procedures. Call depth was limited to 5, and for each test input we also give the computed output value. The test input generation took 1200 ms.

In Table 3, we present the behaviour of our implementation with different procedures, most of them have been chosen from the DPPD library [18]. For each of them, we indicate (1) the mode of the predicate, (2) its determinism, (3) the maximum call depth used, (4) the number of solutions requested (only in the case of non-deterministic and multi-deterministic procedures), (5) the number of test cases generated, and (6) the execution time of the test input generation, given in *ms*.

7 Conclusion and Future Work

In this work we have developed the necessary concepts and machinery for a control-flow based approach to the automatic generation of test inputs for structural testing of Mercury programs. Some parts of our approach, in particular the generation of execution sequences, have been deliberately left general, as it permits to develop algorithms for automatic test input generation that are parametrised with respect to a given coverage criterion or even a strategy for computing a series of "interesting" test inputs.

A side-effect of our approach is that not only test inputs are computed, but also the corresponding *outputs* (the fact that the predicate fails or succeeds and, in the latter case, what output values are produced). All the programmer has to do in order to construct a test suite is then to check whether the generated output corresponds to what is expected.

We have evaluated a prototype implementation that computes a finite set of execution sequences and the associated test inputs but makes no effort whatsoever to guarantee a certain degree of coverage. This is left as a topic for further research. Several improvements can be administered to our prototype in order to improve its performance. For example, one could integrate the constraint solving phase with the execution sequence generation in order to limit the number of generated sequences.

Other topics for further work include the development of a test evaluation framework, that would provide a mechanism for registering automatically generated (and possibly edited) test cases in a form that would allow a repeated evaluation of the test suite when changes in the source code occur (so-called regression testing).

Acknowledgments

The authors would like to thank Baudouin Le Charlier for interesting and productive discussions on the subject of this paper. François Degrave is supported by F.R.I.A. - Fonds pour la formation à la Recherche dans l'Industrie et dans l'Agriculture. Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders.

References

1. Glass, R.: Software runaways: lessons learned from massive software project failures. Prentice-Hall, Englewood Cliffs (1997)
2. Kaner, C., Falk, J., Nguyen, H.Q.: Testing computer software. John Wiley and Sons, Chichester (1993)
3. Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4) (1997)
4. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP 2007: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 63–74. ACM, New York (2007)
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: ICFP 2000: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp. 268–279. ACM, New York (2000)
6. Luo, G., von Bochmann, G., Sarikaya, B., Boyer, M.: Control-flow based testing of prolog programs. In: Proceedings of the Third International Symposium on Software Reliability Engineering, pp. 104–113 (1992)
7. Belli, B., Jack, O.: Implementation-based analysis and testing of prolog programs. In: ISSTA 1993: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, pp. 70–80. ACM Press, New York (1993)
8. Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: Proceedings of ASE 2001 (2001)
9. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA 1998: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pp. 53–62. ACM Press, New York (1998)
10. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Automated Software Engineering, pp. 219–228 (2000)
11. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java pathfinder. SIGSOFT Softw. Eng. Notes 29(4), 97–107 (2004)
12. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 17-19, pp. 365–371 (2004)
13. Somogyi, Z., Henderson, H., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming 29(4) (1997)
14. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for Mercury programs (extended abstract). In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407. Springer, Heidelberg (2007)
15. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence 23, 295–307 (1984)
16. Degrave, F., Vanhoof, W.: A control flow graph for Mercury. In: Proceedings of CICLOPS 2007 (2007)
17. Demoen, B., de la Banda, M.G., Stuckey, P.: Type constraint solving for parametric and ad-hoc polymorphism. In: Edwards, J. (ed.) The 22nd Australian Computer Science Conference, pp. 217–228. Springer, Heidelberg (1999)
18. Leuschel, M.: The dppd library of benchmarks, `http://www.ecs.soton.ac.uk/ mal/systems/dppd.html`