Interactive Multiple Predicate Revision based on Unfolding Transformation

Zoltán Alexin[1], Tibor Gyimóthy[2] and Henrik Boström[3]

[1] Department of Applied Informatics
Attila József University
Árpád tér 2, H-6720 Szeged Hungary
Phone: (36) +(62) 310011 ext. 3818, Fax: (36) +(62) 312292
e-mail: alexin@inf.u-szeged.hu

[2] Research Group on the Theory of Automata
Hungarian Academy of Sciences,
Aradi vértanuk tere 1, H-6720 Szeged Hungary
Phone-fax: (36) +(62) 312508
e-mail: gyimi@inf.u-szeged.hu.

[3] Department of Computer and Systems Sciences
Stockholm University
Electrum 230, 164 40 Kista Sweden
e-mail: henke@dsv.su.se.

Abstract. The algorithm SPECTRE specializes logic programs with respect to positive and negative examples by applying the transformation rule unfolding together with clause removal. The method IMPUT presented in this paper gives a modified version of this algorithm by integrating the algorithmic debugging system IDTS with SPECTRE. The main idea of the IMPUT method is that the identification of a clause to be unfolded has a crucial importance in the effectiveness of the specialization process. The debugging system IDTS is used to identify this *buggy* clause [4].

1 Introduction

In this paper we present a method for the interactive revision of multiple predicates of logic programs. The method (called IMPUT) is based on the interactive debugging technique IDTS introduced in [6] and the specialization algorithm SPECTRE presented in [3]. The main idea of IMPUT is that by combining a debugger with SPECTRE an improvement of the specialization process can be achived.

The IDTS system improves Shapiro's original algorithmic debugging method [20] by reducing the number of questions to the oracle. The validity of results of a procedure call is not asked from the oracle if it can be inferred from a Category Partition Test configuration [12]. Other improvement in IDTS is that

[4] This work is supported by the ESPRIT project BRA 6020 and by the OTKA grant 501.

the only relevant program execution paths that may have effected the value of an incorrect output are analyzed [14].

The problem of specializing of logic program w.r.t positive and negative examples can be viewed as the problem of pruning on SLD-tree so that all refutations of negative examples and no refutations of positive examples are excluded. The actual pruning can be performed by applying unfolding and clause removal. The SPECTRE method is based on this idea. A logic program can be specialized by using different strategies of computation rules, e.g. Prolog, random, impurity measure.

The program revision technique used in the IMPUT system is based on unfolding and clause removal. Since the initial program may contain several predicates this technique includes an efficient interactive debugging method to identify the clause to be revised. This integration of these two components can be more efficient in many cases than the unfolding alone.

The main idea of the IMPUT system is that the identification of a clause to be unfolded has a crucial importance in the effectiveness of the specialization process. If a negative example is covered by the current version of the initial program supposedly there is at least one clause which is responsible for this incorrect covering. The debugging system IDTS is used to identify a buggy clause instance. This clause is removed from the initial program. If a derivation of a positive example contains this clause then the resolvents of the clause are added to the initial program. A modified version of impurity measure strategy is used to determinate the literal to be unfolded.

In Section 2 we briefly describe the SPECTRE and IDTS systems and give some definitions used in the paper. The IMPUT method is discussed in Section 3. Section 4 contains a comparisons to other works. Finally, in Section 5 the summary and comments on future works are given.

2 Preliminaries

In this section we give definitions of some concepts used in this paper and short descriptions of the SPECTRE and IDTS systems.

2.1 Definitions

The specialization of a logic program by unfolding and clause removal is discussed in [3]. If a clause $p_i \in \mathbf{P}$ takes part in the refutation of some negative examples, but it does not occur in the refutations of positive examples, then it may be removed from the program. Such clauses are obtained from an initial program by *unfolding*. Here is the definition of *derivation* and *resolvents* that were introduced by Lloyd in his book [9].

Definition 1. Derivation, resolvents. Let G_i be a goal $\leftarrow A_1, \ldots, A_m, \ldots, A_k$, C_{i+1} be a clause $A \leftarrow B_1, \ldots, B_q$ and R a computation rule. Then G_{i+1} is *derived* from G_i and C_{i+1} using mgu θ_{i+1} via R if the following conditions hold:

- A_m is the selected atom determined by the computation rule R.
- $A_m\theta_{i+1} = A\theta_{i+1}$ (that is θ_{i+1} is an mgu of A_m and A).
- G_{i+1} is the goal $(\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_{i+1}$.

In resolution terminology G_{i+1} is a *resolvents* of G_i and C_{i+1}.

Next we give the formal definition of the *unfolding*.

Definition 2. Unfolding. Let p_i $H \leftarrow A_1, \ldots, A_m, \ldots, A_k$ be a program clause in $\mathbf{P}$, and $C = \{c_1, \ldots, c_q\}$ be a set of program clauses such that the head of each $c_j \in C$ is unifiable with the literal A_m. A_m is selected by some computation rule R. Then the program $\mathbf{P}'$ after unfolding is:

$$\mathbf{P}' = \mathcal{U}(\mathbf{P}) = \mathbf{P} \setminus \{p_i\} \cup \Big(\bigcup_{c_j \in C} H \leftarrow A_1, \ldots, A_{m-1}, body(c_j), A_{m+1}, \ldots, A_k \Big)$$

The clause p_i is replaced by its resolvents in the program $\mathbf{P}$.

Applying the above unfolding operator $\mathcal{U}$ we can obtain new versions of the initial program and after some steps of unfolding we may find clause(s) that can be removed from the program without harming its behavior on the positive examples.

Definition 3. False clause. Let $\mathbf{P}$ be a set of program clauses, and e^- a ground atom. Let the expected program model be $\mathcal{M}$, which is embodied by an *oracle*. ($\mathcal{M}$ may differ from the least Herbrand model M_P.) For the purposes e^- is covered by the program $\mathbf{P}$. Then p_i is said to be a *false clause* in the program $\mathbf{P}$ if the following is hold:

- An instance of p_i *does* occur in the derivation of the goal e^-.
- If the clause instance of p_i that is occurd in the derivation is $B \leftarrow A_1, \ldots, A_k$ and if all atoms in the body of that clause instance are in the model $\mathcal{M}$ but the instance of B **is not** in $\mathcal{M}$.

2.2 The SPECTRE specializing algorithm

The algorithm SPECTRE [3] specializes logic programs with respect to positive and negative examples by applying the transformation rule *unfolding* [21] together with clause removal. This is done in the following way. As long as there is a clause in the program that covers a negative example, it is checked whether it covers any positive examples or not. If it covers no positive examples, then it is removed, otherwise it is unfolded. The choice of which literal to unfold upon is made using a computation rule, which is given as input to the algorithm. The generality of the resulting specialization is dependent on the computation rule, and thus the choice of computation rule is crucial for the performance of the algorithm. The experimental results presented in [3] indicate that the computation rule should be formulated so that the number of clauses that are needed to distinguish between positive and negative examples is minimized. This means that the number of applications of unfolding should be kept as low as possible, since the number of clauses increases when unfolding is applied. In [3] the optimal choice of literal to unfold upon was approximated by selecting the literal that results in the minimal residual impurity of the resolvents when having applied unfolding. The measure of residual impurity used coincides with that is in ID3 [17].

2.3 The IDTS system

The IDTS interactive debugging environment was presented first in [6]. This preliminary version of the IDTS combines the Shapiro's *false procedure* algorithm [20] and the CPM testing method [12]. Later IDTS was augmented with program slicing techniques [14] for making an even more advanced debugging system.

The algorithmic program debugging technique introduced by Shapiro can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly. Shapiro's model was originally applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. A major drawback of this debugging method is the great number of queries to the user about the correctness of intermediate results of procedure calls.

A major improvement in the bug localization process is realized in IDTS by combining the *category partition testing method* [12] with the algorithm introduced in [20]. The main idea is the following: During the debugging of a program the user has to answer many "difficult" questions. If the program has already been tested, the test results for the procedures of the program can be directly applied in the debugging process without consulting the user.

The program slicing part of IDTS is based on the *annotation inference* technique. Using this technique an annotating specification of directionality (*input/output*) can be automatically generated for the functional part of a logic program.[5] The user may annotate more positions according to his/her intended use of the program, and IDTS checks the functional correctness of the final annotation.

From an annotation a *dependence graph* is constructed for the logic program. A proof (refutation) tree is produced for a buggy program and using the dependence graph the tree is sliced, removing those parts that have no influence on the visible symptom of a bug. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program. The annotation

[5] The method has originally been developed for functional logic programs. In IDTS (applied on Prolog) the functional entities are represented by conventional extra–logical primitives, such as arithmetics.

of the program is used for preparing the test database as well: The user may provide test cases over input arguments of the annotated program.

3 The IMPUT system

In the following the IMPUT system is presented. It consists of two major parts. The specialization algorithm comes from the SPECTRE system [3] while the interactive debugger part is imported from the IDTS system. The original specialization algorithm was extended in order to be able to revise multiple predicates simultaneously. This modification initiated some further improvements and these are incorporated in IMPUT as well.

First, the motivation for developing IMPUT system is explained using a simple Prolog program, called `rectangle/4`. The task of this program is to recognize a laying rectangle with vertical left and right sides and with horizontal base and top segments. An initial program and sets of positive and negative examples are presented in Figure 1.

```
rectangle(X,Y,Z,U) :- leftside(X),base(Y), rightside(Z),top(U).
base(X)      :- segment(X).
top(X)       :- segment(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
segment(X) :- horiz(X).
segment(X) :- vert(X).
horiz(h1).
horiz(h2).
horiz(h3).
vert(v1).
vert(v2).
vert(v3).
```

$$\mathbf{E}^{+} = \{\ rectangle(v1,h1,v2,h2), rectangle(v2,h2,v3,h1), rectangle(v1,h3,v1,h3), rectangle(v3,h1,v1,h2)\ \},$$

$$\mathbf{E}^{-} = \{\ rectangle(v1,v2,h1,v2), rectangle(v2,v1,h2,v1), rectangle(v1,h1,v1,v1), rectangle(v2,h1,v2,v2), rectangle(h1,h2,h2,v1), rectangle(h1,v2,h2,h2), rectangle(v1,h2,h2,h3).$$

Fig. 1. The initial program and the examples for `rectangle/4`

Running the original SPECTRE method on this initial program and examples the specialized program is given in Figure 2. This output shows clearly the basic drawback of the SPECTRE method. This algorithm always unfolds the target predicate although in many cases the revision of other clauses would be suitable. Of course, by using more examples the result of SPECTRE system can be improved. However, the expected solution for the program **rectangle/4** could never be achived by this algorithm. The expected solution of the program **rectangle/4** is listed in Figure 3.

The IMPUT system integrating the specialization algorithm of SPECTRE with the interactive debugging system IDTS can infer the expected program for **rectangle/4**. The main idea of IMPUT method is that the identification of the next clause to be unfolded has a crucial importance in the effectiveness of the specialization process. The specialization steps are invoked by the negative examples that are covered by the target predicate.

```
rectangle(v1, h1, A, B) :- segment(A), horiz(B).
rectangle(v1, h3, A, B) :- rightside(A), top(B).
rectangle(v2, h2, A, B) :- rightside(A), top(B).
rectangle(v2, h3, A, B) :- rightside(A), top(B).
rectangle(v3, A, B, C)  :- base(A), rightside(B), top(C).
base(A)      :- segment(A).
top(A)       :- segment(A).
leftside(A)  :- segment(A).
rightside(A) :- segment(A).
segment(A) :- horiz(A).
segment(A) :- vert(A).
horiz(h1).
horiz(h2).
horiz(h3).
vert(v1).
vert(v2).
vert(v3).
```

Fig. 2. The output of the SPECTRE system

We assume, when a negative example is covered by the current version of the program then there is at least one clause which is responsible for this incorrect covering. IMPUT system uses the IDTS debugging algorithm to identify a buggy clause instance of the program. The clause instance identified in this process will be unfolded in the next step of the specialization algorithm.

3.1 The IMPUT algorithm

In the IMPUT method, similarly to the general ILP approach, background knowledge could be given which is not modified during the specialization. In the specialization process there are two approaches for the clause removal. We may either remove as many clauses as possible and get the most specific theory or remove only those clauses whose removal is necessary and get the most general theory. In both approaches the obtained program will succeed on $\mathbf{E}^+$ and will

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- horiz(X).
leftside(X)  :- vert(X).
rightside(X) :- vert(X).
segment(X) :- horiz(X).
segment(X) :- vert(X).
horiz(h1).
horiz(h2).
horiz(h3).
vert(v1).
vert(v2).
vert(v3).
```

Fig. 3. The expected solution of the program `rectangle/4`

fail on $\mathbf{E}^-$. The algorithm presented in the Figure 4 results in the most specific theory, while the other algorithm that results in the most general theory is presented in Figure 5.

For the demonstration we used the algorithm producing the most specific theory. Using the other algorithm gives the same result in this special case, but twice as many steps are needed *(for the clause removal a separate step is needed)*.

In the following we give a detailed description how the IMPUT system is working on the problem listed in Figure 1. In this description we use a very simple single-stepping version of the IDTS without test database and slicing to simplify the explanation of IMPUT method. Later we will discuss the advantage of using CPM test database and slicing during the debugging. Figure 6 shows the initial clauses and the background knowledge of the program `rectangle/4`.

The following result is obtained when applying the IMPUT algorithm:

$\mathbf{i} = 0$.
All positive examples are covered. The first example ($rectangle(v1, v2, h1, v2)$) in $\mathbf{E}^-$ is covered by the program $\mathbf{P}$. Entering into the debugger the following questions are asked (the facts involved in the background knowledge do not invoke a question):

`leftside(v1) is it ok? (y/n)` *y*
`base(v2) is it ok? (y/n)` *n*

By means of the debugger we have found a clause that is responsible for the incorrect behavior. The clause found is specialized by unfolding. Now there is only one literal on the right side of the clause $\texttt{base(X)} : -\texttt{segment(X)}$, therefore the unfolding will be based on that literal. The resolvents of the clause is: $\mathbf{C} = \{\texttt{base(X)} : -\texttt{horiz(X)}, \texttt{base(X)} : -\texttt{vert(X)}\}$. After simplifying $\mathbf{C}$, the

Input: An initial program $\mathbf{P} = \{p_1, \ldots, p_u\}$, background knowledge $\mathbf{B} = \{b_1, \ldots, b_v\}$ (a set of clauses that is not changed during the learning process), sets of ground atoms $\mathbf{E}^+$, $\mathbf{E}^-$ (the positive and the negative examples).
Output: Series of programs $\mathbf{P}$, $\mathbf{P}'$, …, $\mathbf{P}'^{(n)}$ ($\mathbf{P}'^{(0)} = \mathbf{P}$), where $\mathbf{P}'^{(i+1)} = \tilde{\mathcal{U}}\big(\mathbf{P}'^{(i)}\big)$ $(0 \leq i < n)$, $\tilde{\mathcal{U}}$ is the unfolding operator extended with clause removal.

The Algorithm:

Check if the program $\mathbf{P}$ terminates on all $e^+ \in \mathbf{E}^+$.
if fails then stop "Initial program should cover all positive examples."
let $\mathbf{i} = 0$.
while there is an $e^- \in \mathbf{E}^-$ such that $\mathbf{P}'^{(i)}$ does not fail on e^- do
begin
  Find a buggy clause $c \in \mathbf{P}'^{(i)}$ using the IDTS debugger
  Perform unfolding on c using a computation rule.
  let $\mathbf{C} = \{c_1, \ldots, c_s\}$ be the resolvents of c.
  Remove from $\mathbf{C}$ all those clauses that does not occur in
    refutations of positive examples.
  let $\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \ \{c\} \cup \mathbf{C}$
  let $\mathbf{i} = \mathbf{i} + 1$
end

Fig. 4. The IMPUT algorithm

clause `base(X)` $:-$`vert(X)` is removed, because there is no positive example whose derivation contains it. We got the $\mathbf{P}'^{(1)}$, that is listed in Figure 7.

Then $\mathbf{i}$ becomes 1 and the while-cycle continues. This program $\mathbf{P}'^{(1)}$ will not cover the two first negative examples $rectangle(v1, v2, h1, v2)$, $rectangle(v2, v1, h2, v1)$ but will cover the third $rectangle(v1, h1, v1, v1)$. By the debugger we can find an erroneous clause again.

```
base(h1) is it ok? (y/n) y
rightside(v1) is it ok? (y/n) y
top(v1) is it ok? (y/n) n
```

The clause to unfold is: `top(X)` $:-$`segment(X)`. The right side contains only one literal, therefore the unfolding will be based on that literal. The resolvents of the clause is: $\mathbf{C} = \{$`top(X)` $:-$`horiz(X)`, `top(X)` $:-$`vert(X)`$\}$. In the derivation of positive examples the clause `top(X)` $:-$`vert(X)` does not occur, therefore it is removed from $\mathbf{C}$. The result of this unfolding is presented in Figure 8.

Similarly, using the negative example $rectangle(h1, h2, h2, v1)$ we will find that the clause `leftside(X)` $:-$`segment(X)` should be unfolded, then using the negative example $rectangle(v1, h2, h2, h3)$ we will find that the clause `rightside(X)` $:-$`segment(X)` should be unfolded. The program $\mathbf{P}'^{(4)}$ will cover all elements of the $\mathbf{E}^+$ and will fail on all elements of the $\mathbf{E}^-$. The result is the same as the

Check if the program $\mathbf{P}$ terminates on all $e^+ \in \mathbf{E}^+$.
if fails then stop "Initial program should cover all positive examples."
let $\mathbf{i} = 0$.
while there is an $e^- \in \mathbf{E}^-$ such that $\mathbf{P}'^{(i)}$ does not fail on e^- do
begin
  Find a buggy clause $c \in \mathbf{P}'^{(i)}$ using the IDTS debugger
  if c does not occur in the refutations of positive examples
  then Remove c from $\mathbf{P}'^{(i)}$ $\left(\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \{c\}\right)$.
  else begin
    Perform unfolding on c using a computation rule.
    let $\mathbf{C} = \{c_1, \ldots, c_s\}$ denote the resolvents of c.
    let $\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \{c\} \cup \mathbf{C}$
  end
  let $\mathbf{i} = \mathbf{i} + 1$
end

Fig. 5. The IMPUT algorithm for obtaining the most general theory

The initial program is:

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- segment(X).
top(X)       :- segment(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
```

The background knowledge is:

```
segment(X) :- horiz(X).
segment(X) :- vert(X).
horiz(h1).
horiz(h2).
horiz(h3).
vert(v1).
vert(v2).
vert(v3).
```

Fig. 6. The initial program and the background knowledge for the IMPUT system

expected solution listed in Figure 3. The $\mathbf{P}'^{(4)}$ is listed in Figure 9.

A great drawback of the IMPUT system is that the *oracle* has to answer for membership questions to identify a buggy clause instance. However, using the IDTS method the number of these questions may be radically reduced. Applying the slicing component of IDTS only those part of the refutation tree is considered which may effect the incorrect value of an argument in the target predicate. If the predicate `rectangle(X,Y,Z,U)` is invoked by an incorrect value on the argument

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- segment(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
```

Fig. 7. The program $\mathbf{P}'^{(1)}$ after the first unfolding

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- horiz(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
```

Fig. 8. The program $\mathbf{P}'^{(2)}$ after the second unfolding

`U`, then the IDTS system can produce a subtree (a slice) of the refutation tree which contains the nodes derived from predicate `top(X)`. The *oracle* is only asked about the correctness of the nodes of this subtree. (The value of argument `U` may only be affected by these predicates.)

In this paper we demonstrated the IMPUT method using an example which does not contain recursive predicates. The application of SPECTRE method for recursive predicates is discussed in [3].

4 Related Work

In this section first we give a detailed comparison of SPECTRE method with other specialization techniques. After that, we compare the SPECTRE and the IMPUT systems.

In contrast to the specialization techniques in [8, 1], SPECTRE specializes logic programs non-minimally. This means that SPECTRE may exclude other logical consequences of the program than the negative examples. Other non-minimal specialization techniques are described in [20, 2, 13, 16, 4, 11, 19, 22, 5, 15, 23]. They can be divided into three (not disjoint) groups with respect to their specialization techniques: clause removal, addition of literals and goal reduction.

The only specialization operator that is used in [13, 11, 22] is clause removal. In contrast to SPECTRE, the only clauses that are considered for removal in

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- horiz(X).
leftside(X)  :- vert(X).
rightside(X) :- vert(X).
```

Fig. 9. The program $\mathbf{P}'^{(4)}$ which is the result of the IMPUT algorithm

these approaches are clauses that appear in the program to be specialized. As a consequence, these approaches will not be able to produce specializations that include all positive examples when all clauses in a refutation of a negative example are used in refutations of positive examples.

In [23] a technique for incremental specialization is presented, called MBR. It uses clause removal and addition of literals to prevent negative examples from being derived. In contrast to SPECTRE, the aim of this technique is not to specialize a program with respect to positive and negative examples, but make a minimal revision of the program in the sense that a minimal set of clause applications is prevented from being used. It should be noted that SPECTRE is not guaranteed to produce specializations that are minimal in this sense.

In [20, 2, 18, 16, 19, 15], clauses are specialized by adding literals to their bodies. The literals considered for being added are in these approaches restricted to those whose predicate symbols are defined in the original program. Various restrictions are also put on the variables in the literals (e.g. at least one of the variables should appear elsewhere in the clause [18]). The branching factor in the space of possible refinements is normally much larger for these approaches than for SPECTRE. It should be noted that SPECTRE reduces the branching factor at the cost of expressive power, i.e. all specializations produced by SPECTRE can be obtained by the previous approaches, while the contrary does not hold. However, it should also be noted that one refinement step in SPECTRE normally comprise several steps in the previous approaches, since one goal reduction may introduce more than one literal. This means that SPECTRE may find refinements that are missed by techniques that do not search the space exhaustively (e.g. FOIL [18]).

The major difference between SPECTRE and previous techniques that use goal reduction for specializing logic programs (ML-SMART [2], ANA-EBL [4], FOCL [16], GRENDEL [5], FOCL-FRONTIER [15]) is the way in which the search for a specialization is performed. The algorithms ANA-EBL, FOCL, GRENDEL and FOCL-FRONTIER, like MIS [20] and FOIL, use covering methods for finding correct specializations, while SPECTRE uses a divide-and-conquer method. This means that in the previous algorithms the refinement graph is searched repeatedly (cf. the AQ family [10]), while SPECTRE searches the refinement graph once (cf. ID3 [17]). The major difference between ML-SMART and SPECTRE is that ML-SMART only considers reducing the leftmost goal, while SPECTRE reduces the goal that is selected by the computation rule given as input to the algorithm. Experimental results [3] have shown that a dynamic computation rule, which considers the training examples, can give significantly more accurate specializations than the static computation rule used by ML-SMART.

The main difference between SPECTRE and IMPUT system is in the way of selection clauses for unfolding. SPECTRE always unfolds the target predicate. The idea of IMPUT system is that in many cases the unfolding of other predicates would be more useful. When a negative example is covered by the current version of the program supposedly there is at least one clause which is responsible for this incorrect behavior. The IMPUT method identifies a buggy clause and applies the unfolding step to this clause. Hence, IMPUT can be considered a multiple predicate revision tool which contains an interactive debugger to select predicates.

The IMPUT method uses the concept of background knowledge as well. It means, that from the initial program a set of background clauses is separated. The clauses belong to the background are not modified during the specialization process.

5 Conclusion

In this paper a new method for the specialization of logic programs is presented. This method improves the original SPECTRE algorithm by combining it with an interactive debugger to identify a clause for unfolding. This solution has a drawback because an *oracle* has to answer membership questions to identify a buggy clause instance. However, the IDTS method integrated in IMPUT system can reduce effectively the number of these questions. The SPECTRE method and the IDTS system have already been implemented. The implementation of IMPUT system which integrates these two components is currently under development.

References

1. Bain, M., Muggleton, S.: "Non-Monotonic Learning", in Muggleton S. (ed.), *Inductive Logic Programming*, Academic Press, London (1992) 145–161
2. Bergadano, F., Giordana, A.: "A Knowledge Intensive Approach to Concept Induction", *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann, CA (1988) 305–317
3. Boström, H., Idestam-Almquist, P.: Specialization of Logic Programs by Pruning SLD-trees. Proc. of the Fourth International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September 12-14. (1994) 31–47
4. Cohen, W. W.: "The Generality of Overgenerality", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 490–494
5. Cohen, W. W.: "Compiling Prior Knowledge Into an Explicit Bias", *Machine Learning: Proceedings of the Ninth International Workshop*, Morgan Kaufmann (1992) 102–110
6. Horváth, T., Gyimóthy, T., Alexin, T., Kocsis, F.: Interactive Diagnosis and Testing Logic Programs. Proc. of the Third Symposium on Programming Languages and Software Tools, ed. Mati Tombak, Kääriku Estonia August 23-24. (1993) 34–46
7. Kókai, G., Alexin, Z., Kocsis, F.: The IDT System and its Application for Learning Prolog Programs. Proc. of the Sixth International Conference on Artificial Intelligence and Information Control Systems of Robots (AIICSR-94) Smolenice Castle Slovakia September 12-16. (1994) 315–320
8. Ling, C. X.: "Non-Monotonic Specialization", *Proceedings of International Workshop on Inductive Logic Programming*, Portugal (1991) 59–68
9. Lloyd, J. W.: Foundations of Logic Programming, (2nd edition) Springer-Verlag (1987)

10. Michalski, R. S.: "Pattern Recognition as Rule-Guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, bf 2 (1980) 349–361
11. Mooney, R. J., Ourston, D.: "Constructive Induction in Theory Refinement", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 178–182
12. Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM **31:6** June (1988) 676–686
13. Ourston, D., Mooney, R. J.: "Changing the Rules: A Comprehensive Approach to Theory Refinement", *Proceedings of the Eighth National Conference on Artificial Intelligence*, MIT Press (1990) 815–820
14. Paakki, J., Gyimóthy, T., Horváth T.,: Effective Algorithmic Debugging for Inductive Logic Programming. Proc. of the Fourth International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September 12-14. (1994) 175–194
15. Pazzani, M., Brunk, C.: "Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning", *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Morgan Kaufmann (1993) 328–334
16. Pazzani, M., Brunk, C., Silverstein G.: "A Knowledge-Intensive Approach to Learning Relational Concepts", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 432–436
17. Quinlan, J. R.: "Induction of Decision Trees", *Machine Learning* **1** (1986) 81–106
18. Quinlan, J. R.: "Learning Logical Definitions from Relations", *Machine Learning* **5** (1990) 239–266
19. Richards, B. L., Mooney, R. J.: "First-Order Theory Revision", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 447–451
20. Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)
21. Tamaki, H., Sato, T.: "Unfold/Fold Transformations of Logic Programs", *Proceedings of the Second International Logic Programming Conference*, Uppsala University, Uppsala, Sweden (1984) 127–138
22. Wogulis, J.: "Revising Relational Domain Theories", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 462–466
23. Wrobel S.: "On the Proper Definition of Minimality in Specialization and Theory Revision", *Proceedings of the European Conference on Machine Learning*, Springer-Verlag (1993) 65–82