Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner

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

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]

Keywords. Logic and Constraint Programming, Machine Learning

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 [4]. The main idea of IMPUT is that by combining a debugger with SPECTRE an improvement of the specialization process can be achieved.

The IDTS system improves Shapiro's original algorithmic debugging method [12] by reducing the number of questions put 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 [9]. Another improvement in IDTS is that only relevant program execution paths that may affect the value of an incorrect output are analyzed [10].

The problem of specializing a logic program w.r.t positive and negative examples can be viewed as the problem of pruning an 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 algorithm SPECTRE is based on this idea, and it specializes clauses defining a target predicate by using different strategies for selecting the literal to apply unfolding upon (e.g. the leftmost, randomly, using an impurity measure).

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 there is supposedly at least one clause which is responsible for this incorrect covering. The debugging system IDTS is used to identify a buggy clause instance then 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 the impurity measure strategy [4] is used to determinate the literal to be unfolded.

In this paper we use the original SPECTRE algorithm which does not give a direct solution to the problem of specializing recursive predicates. The integration of SPECTRE II [3] (which overcomes this limitation) into the IMPUT system can be done in the same manner.

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, while Section 4 contains a benchmark for IMPUT. Section 5 gives a comparison with other works, and finally in Section 6 a summary and comments on future work are given.

2 Preliminaries

In this section we provide definitions of some of the 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 [4]. Let $\mathbf{P}$ logic program be a set of (program) clauses. 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*. The following definitions of *derivation* and *resolvent* were introduced in [8].

1 Definition (derivation, resolvent)

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.
- $A_m\theta_{i+1} = A\theta_{i+1}$.
- 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 *resolvent* of G_i and C_{i+1}.

2 Definition (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

[1] Department of Applied Informatics József Attila University Árpád tér 2, H-6720 Szeged Hungary Phone: (36) +(62) 454293, Fax: (36) +(62) 312292 e-mail: alexin@inf.u-szeged.hu

[2] Research Group on Artificial Intelligence of the Hungarian Academy of Sciences, Aradi vértanuk tere 1, H-6720 Szeged Hungary Phone: (36) +(62) 454139, 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

[4] This work is supported by grants PHARE TDQM 9305-02/1022 and OTKA T14228.

© 1996 Z. Alexin, T. Gyimóthy and H. Boström
ECAI 96. *12th European Conference on Artificial Intelligence*
Edited by W. Wahlster
Published in 1996 by John Wiley & Sons, Ltd.

each $c_j \in C$ is unifiable with the literal A_m by some θ_j unifier. A_m is selected by some computation rule R. Then the program $\mathbf{P}'$ after unfolding is:

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

where 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.

3 Definition (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 holds:

- An instance of p_i *does* occur in the derivation of the goal e^-.
- If the clause instance of p_i that occurs 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 B **is not** in $\mathcal{M}$.

2.2 The algorithm SPECTRE

The algorithm SPECTRE [4] specializes logic programs with respect to positive and negative examples by applying the transformation rule *unfolding* [13] 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 [4] 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 [4] 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 used in ID3 [11].

2.3 The IDTS system

The IDTS interactive debugging environment was first presented in [6]. This preliminary version of the IDTS combines the Shapiro's *false procedure* algorithm [12] and the CPM testing method [9]. Later IDTS was augmented with program slicing techniques [10] for making an even more advanced debugging system.

The algorithmic program debugging method 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 made 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* [9] with the algorithm introduced in [12]. The main idea is as follows: 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 category partition testing part of the IDTS can be used if an initial test database has been defined. As a drawback, this initial test configuration can be considered as an extra knowledge for the debugger (and for the incremental learner). However, in the learning process we usually have a preliminary assumption about the expected behavior of a predicate to be learned. From this point of view, a category partition test specification can be seen as a higher-order description of the program, with a close resemblance to *integrity constraints* [5].

The program slicing part of IDTS is based on the *annotation inference*[5] technique. Using this technique an annotating specification of directionality (*input, output*) can be automatically generated for the functional part of a logic program. The user may annotate more positions according to his/her intended use of the program, and IDTS will check 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 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 main parts. The specialization algorithm comes from the SPECTRE system [4] 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 [7].

First, the motivation for developing the IMPUT system is explained using a simple Prolog program, called `rectangle/4`. Altough this example is not very realistic, we use it for the clear demonstration of the IMPUT method. In Section 4 results of IMPUT are presented for more realistic domains. The task of this program is to recognize a horizontally lying 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 [6].

Running SPECTRE on this initial program and examples results in the program in Figure 2. This output clearly shows the basic drawback

[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.

[6] In the examples below the `line(X)` denotes a line segment with slope `X`; `line(inf)` denotes a vertical line with infinte slope.

```
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(line(X))   :-  Z is abs(X), Z < 0.01.
vert(line(X))    :-  Z is abs(X), Z > 100.
```

$\mathbf{E}^+ = \{$
$rectangle(line(150), line(0.005), line(inf), line(0.0)),$
$rectangle(line(-160), line(0.0), line(inf), line(0.005)),$
$rectangle(line(150), line(-0.004), line(150), line(-0.004)),$
$rectangle(line(inf), line(0.005), line(150), line(0.0))\},$
$\mathbf{E}^- = \{$
$rectangle(line(150), line(-160), line(0.005), line(-160)),$
$rectangle(line(-170), line(160), line(0.0), line(180)),$
$rectangle(line(200), line(0.001), line(300), line(400)),$
$rectangle(line(-160), line(0.005), line(-160), line(-160)),$
$rectangle(line(0.005), line(0.0), line(0.0), line(150)),$
$rectangle(line(0.005), line(-160), line(0.0), line(0.0)),$
$rectangle(line(150), line(0.0), line(0.0), line(-0.004))\}.$

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

of SPECTRE. This algorithm always unfolds clauses defining the target predicate although in many cases the revision of other clauses would be more appropriate. Of course, by utilizing more examples the result of SPECTRE system can be improved. However, the expected solution for the program `rectangle/4` can never be achieved 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 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.

We assume, when a negative example is covered by the current version of the program, that there is at least one clause which is responsible for this incorrect covering. IMPUT uses the IDTS debugging algorithm to identify a buggy clause of the program. The clause 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 can be given which is not modified during the specialization. In the specialization process there are two approaches for the clause removal. Either one removes as many clauses as possible and get a *most specific* theory or one removes only those clauses whose removal is necessary and get a *least general* theory. In both approaches the obtained program will succeed on $\mathbf{E}^+$ and fail on $\mathbf{E}^-$. The algorithm presented in the Figure 4 results in a *most specific* theory.

For demonstration purposes we used the algorithm producing a *most specific* theory. The other algorithm gives the same result in this

```
rectangle  (line(150), line(0.005), A, B)
           :- segment(A), horiz(B).
rectangle  (line(150), line(-0.004), A, B)
           :- rightside(A), top(B).
rectangle  (line(-160), line(0.0), A, B)
           :- rightside(A), top(B).
rectangle  (line(-160), line(-0.004), A, B)
           :- rightside(A), top(B).
rectangle  (line(inf), 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(line(X))   :-  Z is abs(X), Z < 0.01.
vert(line(X))    :-  Z is abs(X), Z > 100.
```

Figure 2. The output of the SPECTRE system

```
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(line(X))   :-  Z is abs(X), Z < 0.01.
vert(line(X))    :-  Z is abs(X), Z > 100.
```

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

special case, but twice as many steps are needed *(a separate step is needed for the clause removal)*.

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 IDTS with a test database but without slicing to simplify the explanation of IMPUT. Later we will discuss the advantage of using slicing during the debugging process. Figure 5 shows the initial clauses and the background knowledge of the program `rectangle/4`.

Running IMPUT on the example:

Suppose we have the following CPM test specification of the predicate `segment(X)`:

Category Measure of the slope

Choice small { abs(slope) $<$ 0.01 }
Choice medium { 0.01 $\leq$ abs(slope) $<$ 100 }
Choice big { abs(slope) $>$ 100 }
Choice zero { slope = 0 } **property** zero
Choice inf { slope = inf } **property** infinite

Category Sign of the slope

Choice negative **if** not zero and not inf
Choice positive

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 $\mathbf{P}'^{(0)}, \mathbf{P}'^{(1)}, \ldots, \mathbf{P}'^{(n)}$ ($\mathbf{P}'^{(0)} = \mathbf{P}$), where $\mathbf{P}'^{(i+1)} = \tilde{\mathcal{U}}\big(\mathbf{P}'^{(i)}\big)$ $(0 \le i < n)$, $\tilde{\mathcal{U}}$ 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 do 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

Figure 4. The IMPUT algorithm for most specific 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).
segment(X)     :-  horiz(X).
segment(X)     :-  vert(X).
```

The background knowledge is:

```
horiz(line(X))   :-  Z is abs(X), Z < 0.01.
vert(line(X))    :-  Z is abs(X), Z > 100.
```

Figure 5. The initial program and the background knowledge for the IMPUT system

From this test specification IDTS generates the following test frames:

{*(small, negative), (small, positive), (medium, negative), (medium, positive), (big, negative), (big, positive), (zero, positive), (inf, positive)*}

It is assumed that items corresponding to horizontal segments have been tested with result *correct* and no testing has been performed on other items.
Let us apply the IMPUT algorithm listed in Figure 4.

$\mathbf{i} = 0$.
All positive examples are covered. The first example *rectangle(line(150), line(-160), line(0.005), line(-160))* 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 and those having a correct evaluation answer in the test database do not invoke a question):

```
segment(line(150)) is it ok? (y/n) y
leftside(line(150)) is it ok? (y/n) y
segment(line(-160)) is it ok? (y/n) y
base(line(-160)) is it ok? (y/n) n
```

By means of the debugger a clause was found 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 $\mathtt{base(X)} : -\mathtt{segment(X)}$, therefore the unfolding will be based on that literal. The resolvents of the clause are: $\mathbf{C} = \{\mathtt{base(X)} : -\mathtt{horiz(X)}, \mathtt{base(X)} : -\mathtt{vert(X)}\}$. After simplifying $\mathbf{C}$, the clause $\mathtt{base(X)} : -\mathtt{vert(X)}$ is removed, because there is no positive example whose derivation contains it. We obtained the $\mathbf{P}'^{(1)}$, which is listed in Figure 6.

```
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).
segment(X)     :-  horiz(X).
segment(X)     :-  vert(X).
```

Figure 6. The program $\mathbf{P}'^{(1)}$ after the first unfolding

Using the IMPUT algorithm after three further steps we obtained the expected solution listed in Figure 3.

A great drawback of the IMPUT system is that the *oracle* has to answer membership questions to identify a buggy clause instance. However, using the IDTS method the number of these questions can be radically reduced. When applying the slicing component of IDTS, only that 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 `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.)

4 Experimental Results

The IMPUT algorithm was tested on three domains. The number of clauses and the accuracy of the resulting theory were examined. The test domains were the following: *rectangle*, *sentence*, *shuttle*. The *shuttle* domain was used in [4] for testing the SPECTRE algorithm while the other domains were newly added.

We used the same testing technique as in [4] i.e. the sets of positive and negative examples were randomly cut into two parts. One part was used for learning and the other was used for testing. We ran it several times and computed the average accuracy and the average number of clauses in the resulting theory. The results are summarized in the following three tables.

IMPUT was tested using two different computation rules: prolog (i.e. choosing the leftmost literal), and using an impurity measure. For the sake of comparison we ran the SPECTRE algorithm with the impurity measure computation rule on the same examples.

The results of the benchmarks using: IMPUT algorithm with Prolog computation rule				
The domain	Number of clauses	Accuracy		
		Positive	Negative	Overall
rectangle	5.0	100.00%	84.62%	92.21%
sentence	5.5	46.34%	85.00%	65.43%
shuttle	18.6	65.58%	75.58%	70.81%

The results of the benchmarks using: SPECTRE algorithm with impurity measure				
The domain	Number of clauses	Accuracy		
		Positive	Negative	Overall
rectangle	7.8	73.68%	46.15%	59.74%
sentence	10.2	92.68%	80.00%	86.42%
shuttle	5.1	99.64%	96.37%	97.93%

The results of the benchmarks using: IMPUT algorithm with impurity measure				
The domain	Number of clauses	Accuracy		
		Positive	Negative	Overall
rectangle	5.0	100.00%	84.62%	92.21%
sentence	5.9	73.17%	92.50%	82.71%
shuttle	4.0	99.64%	97.69%	98.62%

From these results it can be concluded that using the impurity measure as computation rule gives the best solution w.r.t the number of clauses and accuracy. Comparing IMPUT and SPECTRE we see that the number of clauses learned by IMPUT is less than that learned by the SPECTRE technique. It means that IMPUT can learn more compact theories than SPECTRE due to the *extra knowledge* it has, either stored in the CPM test database or entered by the *oracle*. In most cases we achieved better accuracy with the IMPUT system than with SPECTRE.

5 Related Work

In this section we first compare IMPUT and SPECTRE, then we compare IMPUT with other systems.

The main difference between SPECTRE and IMPUT is the way in which clauses are selected for unfolding. SPECTRE always unfolds a clause that defines the target predicate. The idea of IMPUT is that in many cases it is more appropriate to apply unfolding upon clauses defining other predicates than the target predicate.

When a negative example is covered by the current version of the program, there is supposedly 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.

Like IMPUT, the algorithm SPECTRE II [3] can also be used to specialize other predicates than the target predicate. However, in contrast to IMPUT, the choice of which clause to apply unfolding upon is made non-deterministically in SPECTRE II, while IMPUT uses the IDTS system to select a clause. The major advantage of SPECTRE II over SPECTRE is its ability to produce recursive specializations.

The theory revision system JIGSAW [1] is similar to IMPUT in that it is an integration of an existing theory revisor (RUTH [2]) and SPECTRE. The main difference between JIGSAW and IMPUT is that the latter uses an interactive algorithm to identify *buggy* clauses for applying unfolding upon, while the former uses a non-interactive depth-first iterative deepening scheme for finding how to minimally revise the original theory.

6 Conclusion

In this paper a new method called IMPUT is presented, for the specialization of logic programs. This method improves the original SPECTRE algorithm by combining it with an interactive debugger to identify a clause for unfolding. This solution has one big drawback that an *oracle* has to answer membership questions to identify a buggy clause instance. However, the IDTS method integrated in IMPUT can effectively reduce the number of these questions. One drawback of the IDTS method is that the initial test configuration can be considered as *extra knowledge* for the algorithmic debugger, although we usually have a preliminary assumption about the expected behavior of the predicate to be learned. From this point of view, a category partition specification can be seen as a higher-order description of the program with a close resemblance to integrity constraints [5].

The IMPUT system which integrates SPECTRE and IDTS methods has been fully implemented, while the integration of SPECTRE II into IMPUT is currently under development.

REFERENCES

[1] Adé, H., Boström, H.: JIGSAW: puzzling together RUTH and SPECTRE *Proceedings of the Eighth European Conference on Machine Learning*, Springer-Verlag (1995)

[2] Adé, H., Malfait, B., De Raedt, L.: RUTH : an ILP Theory Revision System. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS94)*, 1994.

[3] Boström, H.: "Specialization of Recursive Predicates", *Proceedings of the Eighth European Conference on Machine Learning*, Springer-Verlag (1995) 92–106

[4] 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

[5] De Raedt, L., Bruynooghe, M.: "Belief Updating from Integrity Constraints and Queries" *Artificial Intelligence*, 53, 2–3, (1992) 291–307

[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] Lloyd, J. W.: Foundations of Logic Programming, (2nd edition) Springer-Verlag (1987)

[9] Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM **31:6** June (1988) 676–686

[10] 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

[11] Quinlan, J. R.: "Induction of Decision Trees", *Machine Learning* **1** (1986) 81–106

[12] Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)

[13] 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