IMPUT: An Iteractive Learning Tool based on Program Specialization

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

[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

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 on the effectiveness of the specialization process. The debugging system IDTS is used to identify this *buggy* clause [4].

Keywords: Inductive Logic Programming, Program Specialization, Algorithmic Debugging

1 Introduction

The problem of specializing a logic program with respect to 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 an unfolding or a clause removal step. The SPECTRE algorithm [7] 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. taking the leftmost, selecting randomly or using the impurity measure).

[4] This work is supported by grants PHARE TDQM 9305-02/1022 and ESPRIT LTR 20237.

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 IDTS interactive debugging technique introduced in [24][13] and the specialization algorithm called SPECTRE. The main idea behind 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 [31] by reducing the number of questions that must be 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 [22]. Another advantage of IDTS is that only relevant program execution paths which may affect the value of an incorrect output are analyzed [24].

The essence of the IMPUT system is that the identification of a clause to be unfolded has a crucial importance on 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 that is responsible for this incorrect covering. The debugging system IDTS is used to identify a buggy clause instance which is then 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 [7] is used to determine the literal to be unfolded.

One special class of Prolog programs is the class of *parsers* for formal languages. The most often used tool for writing parsers is the Definite Clause Grammar (DCG) formalism [27]. DCG enables using not only formal grammar rules within a Prolog program but variables (attributes) in the rules as well. The IMPUT system is capable of specializing DCG rules, therefore it makes possible to learn natural and other (sub)grammars for example the ECG waveform grammar or other biomedical waveform grammars [15].

The working of the IMPUT system is demonstrated by means of examples of different complexity, beginning from a simple one and finally arriving at the PECG application, which is an ECG waveform classifier extended with a learning module. In Section 2 we briefly describe the IDTS and SPECTRE systems and state some definitions made use of in this paper. Afterwards the IMPUT method itself is discussed in Section 3. In Section 4 we present an example, that demonstrates how IMPUT can be used to infer DCG programs. Section 5 contains some benchmarks on SPECTRE and IMPUT. In Section 6 a more involved application of IMPUT, that of the extended PECG system is presented. Section 7 gives an account of related works, and some interesting conclusions are mentioned in Section 8. In Appendix A and B two examples on IMPUT sessions are given along with computer printouts to aid the readers' understanding.

2 Preliminaries

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

2.1 Definitions

The specialization of a logic program by unfolding and clause removal is discussed in [7]. 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 [18].

Definition 1 (derivation, resolvent) Let G_i be a goal clause $\leftarrow A_1, \ldots, A_m, \ldots, A_k$, C_{i+1} be a program 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}.

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 by some unifier θ_j. 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.

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 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 Category Partition Testing Method

The original CPM method is developed for imperative languages [22]. The reformulation of this method for Prolog and its formal description can be found in [24][13].

During the process of functional testing, the programs (procedures) cannot be tested with all possible properties of the input parameters. Hence, the tester's first task is to define the critical properties of parameters. These critical properties are called *categories*.

The categories are divided into classes (called *choices*) presuming that the behavior of the elements of one choice is identical from the point of view of the testing.

If the categories and choices for a program have been defined, then all the possible *test frames* can be generated. A test frame contains exactly one choice from each category. In general, there are many superfluous frames among the generated test frames. These frames can be eliminated by associating *selector expressions* with the choices. A choice can be made in a test frame if the selector expression associated with the choice is true. The selector expressions contain *property names*. A property name is also associated with a choice and can be considered as a logical variable. The value of this variable is true if the given frame contains that choice.

The below example shows a test specification for the `line(X)` Prolog predicate. In the examples below the `line(X)` denotes a line segment with slope `X`; `line(inf)` denotes a vertical line with infinite slope. Two categories are defined: the *Measure of the slope* and the *Sign of the slope*. In each category we have several choices. The two properties: *zero* and *infinite* help to reduce the number of generated test frames.

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

From the above test specification the following test frames can be generated:

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

2.3 The IDTS System

The IDTS interactive debugging environment was first presented in [24][13]. This preliminary version of the IDTS combines the Shapiro's *false procedure* algorithm [31] (see Figure 1) and the CPM testing method [22]. Currently, IDTS also works with the *divide-and-query* [31] and the *top-down* debugging algorithms as well. Later IDTS was augmented with program slicing techniques [24] 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

```
fp((A,B),X) :- fp(A, Y), (Y == ok -> fp(B,X); X = Y).
    fp(A,X) :- issystem(A) -> (A), X = ok ;
               clause(A, B), fp(B, Y),
               (Y ≐ ok -> X = Y ;
               query(forall, A, true) -> X = ok; X = (A:-B)).
```

Fig. 1. Shapiro's *false-procedure* algorithm

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* [22] with the algorithm introduced in [31]. 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* [11].

The program slicing part of IDTS is based on the *annotation inference*[5] technique introduced in [8]. 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 [12]. 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.

2.4 The SPECTRE Algorithm

The algorithm SPECTRE [7] specializes logic programs with respect to positive and negative examples by applying the transformation rule *unfolding* [34] 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

[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.

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 [7] 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 [7] 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 [28].

3 The IMPUT System

IMPUT is an abductive learning algorithm that applies *unfolding* and *clause removal* transformations to the input Prolog program taking into account the sets of positive and negative examples. At each step of the algorithm we get *better and better* programs in the sense that the intermediate hypothesises still refute on all of the positive examples but will fail on more negative examples. When the algorithm terminates, the resulted program will fail on all of the negative examples. Unfortunately, the termination of the algorithm is not guaranteed. Cheng and Wolf [21] discussed its theoretical background and proposed a complete specialization algorithm that will always terminate and can obtain a correct solution for any possible input.
The IMPUT algorithm consists of three steps:

- finding that clause, the unfolding is applied upon — this step is done by the IDTS debugger
- finding the literal within that clause, that will be the basis of the unfolding — the SPECTRE algorithm is used here [7]
- performing the unfolding on the program

The working of IMPUT is demonstrated by the help of the `rectangle/4` example shown in Figure 2. 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. One possible solution is presented in Figure 3.

For a comparison SPECTRE is also tested with this example. The obtained result can be seen in Figure 4. The output clearly shows the basic drawback of the SPECTRE. It 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 the SPECTRE system can be improved. However, such solution that is presented in Figure 3 can never be achieved by this algorithm, although SPECTRE can produce an equivalent program.

On the other hand, the IMPUT system integrating the specialization algorithm of SPECTRE with the IDTS interactive debugging system can infer the expected program for `rectangle/4`.

```
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.
```

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

Fig. 2. The initial program and the examples for `rectangle/4`

```
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.
```

Fig. 3. One possible solution for the `rectangle/4` program

3.1 The IMPUT algorithm

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. It is assumed that, when a negative example is covered by the current hypothesis (intermediate program), then 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.

```
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.
```

Fig. 4. The output of the SPECTRE system with the *impurity measure* computation rule

In the IMPUT method, similarly to the general ILP approach, background knowledge can also be given which is not modified during the specialization. This means that the input clauses are labeled and those clauses belong to the background are not debugged by the IDTS, therefore will not be unfolded. Background clauses remain *unchanged* during the specialization process.

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 specific* theory. In both approaches the obtained program will succeed on $\mathbf{E}^+$ and will fail on $\mathbf{E}^-$. The algorithm presented in Figure 5 results in a most specific theory. The other one, presented in Figure 6 results in a least specific theory.

For demonstration purposes we used the algorithm produces a *most specific* theory. The *least specific theory* algorithm gives the same result in this special case, but it needs twice as many steps (*a separate step is needed for the clause removal*).

3.2 Running IMPUT on the `rectangle` example:

In the following we give a detailed description of how the IMPUT system is working on the `rectangle/4` example listed in Figure 2. This example also shows the usage of test database during the debugging. The `horiz/1` and `vert/1` predicates are labeled as background clauses. It will reduce the number of questions during the debugging. Figure 7 shows the initial clauses and the background knowledge for the program `rectangle/4`.
The following CPM test specification for the predicate `segment/1` is given:

Category Measure of the slope

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 $\mathbf{P}'^{(0)}$, $\mathbf{P}'^{(1)}$, …, $\mathbf{P}'^{(n)}$ ($\mathbf{P}'^{(0)} = \mathbf{P}$), 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:

if the program $\mathbf{P}$ does not terminate on all $e^+ \in \mathbf{E}^+$.
then stop "Initial program should cover all positive examples."
let i $= 0$.
while there is an $e^- \in \mathbf{E}^-$ such that $\mathbf{P}'^{(i)}$ does not fail on e^- do
begin
  Find the responsible clause $c \in \mathbf{P}'^{(i)}$ using the IDTS debugger
  (such clause should exist)
  Select the more suitable literal in the body of c on which the unfolding will be applied
  if such literal does not exist **then** abort.
  Perform unfolding on c.
  let 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 i $=$ **i** $+ 1$
end

Fig. 5. The IMPUT algorithm for most specific theory

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

From the above test specification IDTS generates the following test frames:

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

The test database contains test cases for a given predicate and a test frame. A test case consists of: input arguments; the result of running the the program with these arguments (it is a logical value true/fail); and the evaluation of this test case by the *oracle*. The evaluation can be *undefined*, *correct* or *incorrect*. The test database is represented by a set of Prolog facts. Each fact corresponds to a predicate and one of its test frame. The input arguments for the test case, the result and the evaluation of the test frame (that

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:

if the program $\mathbf{P}$ does not terminate on all $e^+ \in \mathbf{E}^+$.
then stop "Initial program should cover all positive examples."
let i $= 0$.
while there is an $e^- \in \mathbf{E}^-$ such that $\mathbf{P}'^{(i)}$ does not fail on e^- do
begin
  Find the buggy clause $c \in \mathbf{P}'^{(i)}$ using the IDTS debugger
  (such clause should exist)
  if c does not occur in the refutations of positive examples
  then Remove c from $\mathbf{P}'^{(i)}$ $\big(\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \{c\}\big)$.
  else begin
    Select the more suitable literal in the body of c on which the unfolding will be applied
    if such literal does not exist **then** abort.
    Perform unfolding on c.
    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 i $=$ **i** $+ 1$
end

Fig. 6. The IMPUT algorithm for least specific theory

is aggregated from the evaluations of test cases belonging to this test frame) are stored as arguments of `test/6` predicate. The user may enter evaluations to some (or all) test frames when beginning the IMPUT session. Let us suppose that the test database contains the following items:

```
test(segment,1,[small,negative],[line(-0.001)],true,correct).
test(segment,1,[small,positive],[line(0.002)],true,correct).
test(segment,1,[medium,negative], _, _, undefined).
test(segment,1,[medium,positive], _, _, undefined).
test(segment,1,[big,negative], _, _, undefined).
test(segment,1,[big,positive], _, _, undefined).
test(segment,1,[zero,positive],[line(0)],true,correct).
test(segment,1,[inf,positive], _, _, undefined).
```

The above test database corresponds to that the horizontal segments have been tested with result *correct*. The other test frames were left *undefined*. Let us apply the IMPUT algorithm listed in Figure 5.

i $= 0$.

All positive examples are covered. The first example *rectangle(line(150), line(-160),* 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.
```

Fig. 7. The initial program and the background knowledge for the IMPUT system

```
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).
```

Fig. 8. The program $\mathbf{P}'^{(1)}$ after the first step of IMPUT

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 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
```

Let us follow the steps of the IMPUT algorithm listed in Figure 5. By means of the debugger a clause has been found that is responsible for the incorrect behavior. The clause is then being specialized by unfolding. There is only one literal on the right side of the $\texttt{base}(\texttt{X}) : -\texttt{segment}(\texttt{X})$ clause, therefore the unfolding will be based on that literal. The resolvents of the clause are: $\mathbf{C} = \{\texttt{base}(\texttt{X}) : -\texttt{horiz}(\texttt{X}), \texttt{base}(\texttt{X}) : -\texttt{vert}(\texttt{X})\}$. The clause $\texttt{base}(\texttt{X}) : -\texttt{vert}(\texttt{X})$ is removed from $\mathbf{C}$, because there is no positive example whose derivation contains it. The first intermediate hypothesis for `rectangle/4` $\mathbf{P}'^{(1)}$ is listed in Figure 8.

$\mathbf{i} = 1$.

Before the second step the test database will contain the following items:

```
test(segment,1,[small,negative],[line(-0.001)],true,correct).
test(segment,1,[small,positive],[line(0.002)],true,correct).
test(segment,1,[medium,negative], _, _, undefined).
test(segment,1,[medium,positive], _, _, undefined).
test(segment,1,[big,negative],[line(-160)],true,correct).
test(segment,1,[big,positive],[line(150)],true,correct).
test(segment,1,[zero,positive],[line(0)],true,correct).
test(segment,1,[inf,positive], _, _, undefined).
```

Two test frames were changed in the test database: *(big, positive)* and *(big, negative)*. Let us understand that after the test database filled so, the IDTS debugger will not ask facts on `segment/1` predicate any longer excluding the test frames *(medium, negative)*, *(medium, positive)* and *(inf, positive)*. IDTS also remembers the sporadic facts that was evaluated by the user but not stored into the test database.

The program $\mathbf{P}'^{(1)}$ will not cover the first, second and sixth negative examples:

```
rectangle(line(150),line(-160),line(0.005),line(-160)),
rectangle(line(-170),line(160),line(0.0),line(180)),
rectangle(line(0.005),line(-160),line(0.0),line(0.0)).
```

but will cover the third negative example:

```
rectangle(line(200),line(0.001),line(300),line(400)).
```

Entering into the debugger the following questions are asked:

leftside(line(150)) is it ok? (y/n) y
this fact is stored in the debugger — not asked
`base(line(0.001)) is it ok? (y/n)` *y*
`rightside(line(300)) is it ok? (y/n)` *y*
`top(line(400)) is it ok? (y/n)` *n*

The clause to unfold is: $\texttt{top(X)} : -\texttt{segment(X)}$. The right side contains only one literal, therefore the unfolding will be based on that literal. The resolvents of the clause is: $\mathbf{C} = \{\texttt{top(X)} : -\texttt{horiz(X)}, \texttt{top(X)} : -\texttt{vert(X)}\}$. The $\texttt{top(X)} : -\texttt{vert(X)}$ clause does not occur in the derivation of positive examples therefore it is removed from $\mathbf{C}$. The result of the second step is presented in Figure 9.

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

Fig. 9. The program $\mathbf{P}'^{(2)}$ after the second step of IMPUT

$\mathbf{i} = 2.$

Then, using the negative example *rectangle(line(0.005), line(0.0), line(0.0), line(0.005))* we will find that the clause $\mathtt{leftside(X) :-segment(X)}$ should be unfolded. Similarly to the previous steps $\mathbf{P}'^{(3)}$ is obtained. Finally only one negative examples left covered by the program $\mathbf{P}'^{(3)}$: *rectangle(line(150), line(0.0), line(0.0), line(-0.004))*. The $\mathtt{rightside(X) :-segment(X)}$ clause should be unfolded. The resulted program $\mathbf{P}'^{(4)}$ — see in Figure 10 — 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 expected solution listed in Figure 3.

```
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).
```

Fig. 10. The program $\mathbf{P}'^{(4)}$ which is the result of the IMPUT algorithm

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 the number of these questions can be radically reduced.

4 The Sentence Example

In this section we will present the `sentence` example. The sentence program is a parser for a very simple English sentence grammar. See Figure 11. The program is given in DCG [27] that means the program is given in set of formal grammar rules instead of Prolog clauses.

The `sentence` program recognizes English sentences having a simple *verb-phrase* and a simple *noun-phrase*. Both of them may be singular and plural. For the purposes, the initial program does not match the phrases to each other. The example shows how IMPUT can learn from the examples the correct version of it, that will match the singular verb-phrase to singular noun-phrase and the plural verb-phrase to the plural noun-phrase.

The `sentence` example contains seven positive and three negative examples. The last four positive examples are technical ones. These facts enforce IMPUT to find the solution by unfolding instead of deletion clauses for predicates `nounphrase` and `verbphrase`.

Let us start with the negative example: *"The boys plays"*. The IDTS debugger will find that the responsible clause is $\mathtt{sentence} \rightarrow \mathtt{nounphrase}, \mathtt{verbphrase}$. IMPUT will unfold the second literal and results in the

```
0: sentence-->nounphrase, verbphrase.
1: nounphrase-->singular_nounphrase.
2: nounphrase-->plural_nounphrase.
3: singular_nounphrase-->nm.
4: singular_nounphrase-->singular_determiner, singular_noun.
5: plural_nounphrase-->plural_determiner, plural_noun.
6: plural_nounphrase-->plural_noun.
7: verbphrase-->singular_verb.
8: verbphrase-->plural_verb.
```

$\mathbf{E}^{+}= \{$
$sentence([bob, plays], []),$
$sentence([dogs, run], []),$
$sentence([the, boy, jumps], []),$
$nounphrase([bob], []),$
$nounphrase([the, dogs], []),$
$verbphrase([run], []),$
$verbphrase([runs], [])\}$
$\mathbf{E}^{-}= \{$
$sentence([the, boys, plays], []),$
$sentence([boys, runs], []),$
$sentence([a, dog, jump], [])\}$

Fig. 11. The `sentence` example

$$\mathbf{C} = \{\ \mathtt{sentence} \rightarrow \mathtt{singular\_nounphrase}, \mathtt{verbphrase} \\ \qquad \mathtt{sentence} \rightarrow \mathtt{plural\_nounphrase}, \mathtt{verbphrase}\}$$

resolvents. IMPUT will remove the *false-clause* and adds the resolvents **C** to the program. At this time IMPUT can not remove any clauses from the program, because of the positive examples. The program after the first step is presented in Figure 12.

```
0-1: sentence-->singular_nounphrase, verbphrase.
0-2: sentence-->plural_nounphrase, verbphrase.
1: nounphrase-->singular_nounphrase.
2: nounphrase-->plural_nounphrase.
3: singular_nounphrase-->nm.
4: singular_nounphrase-->singular_determiner, singular_noun.
5: plural_nounphrase-->plural_determiner, plural_noun.
6: plural_nounphrase-->plural_noun.
7: verbphrase-->singular_verb.
8: verbphrase-->plural_verb.
```

Fig. 12. The `sentence` program after the first unfolding step

For the better understanding in IMPUT all clauses are numbered. Clauses obtained by unfolding are denoted by pairs instead of single numbers. The first element in the pair is the number of the original clause, the second is the number of the substituted clause. (The literal position where the unfolding was done is not stored — mostly it is obvious.) The clause number `0-1` denotes the clause that was obtained from the clause number `0` by substituting the clause number `1` into it.

```
0-1: sentence-->singular_nounphrase, verbphrase.
0-2-8: sentence-->plural_nounphrase, plural_verb.
1: nounphrase-->singular_nounphrase.
2: nounphrase-->plural_nounphrase.
3: singular_nounphrase-->nm.
4: singular_nounphrase-->singular_determiner, singular_noun.
5: plural_nounphrase-->plural_determiner, plural_noun.
6: plural_nounphrase-->plural_noun.
7: verbphrase-->singular_verb.
8: verbphrase-->plural_verb.
```

Fig. 13. The `sentence` program after the second unfolding step

The program obtained by unfolding will always be logically equivalent with the original one if no one clause is deleted. This is what happened now. The program presented in Figure 12 will cover the first negative example *"The boys plays"*. Cover means that the `sentence` program refutes this example. The responsible clause for it is $\texttt{sentence} \rightarrow \texttt{plural\_nounphrase}, \texttt{verbphrase}$. IMPUT will unfold the first literal and results in the $\mathbf{C} = \{\texttt{sentence} \rightarrow \texttt{plural\_nounphrase}, \texttt{singular\_verb}$ and $\texttt{sentence} \rightarrow \texttt{plural\_nounphrase}, \texttt{plural\_verb}\}$ resolvents. The first clause does not occur is the derivation of positive examples, therefore IMPUT removes it from $\mathbf{C}$. IMPUT also removes the *false-clause* from the original program and adds $\mathbf{C}$ to the program. The program after the second step is presented in Figure 13.

```
0-1-7: sentence-->singular_nounphrase, singular_verb.
0-2-8: sentence-->plural_nounphrase, plural_verb.
1: nounphrase-->singular_nounphrase.
2: nounphrase-->plural_nounphrase.
3: singular_nounphrase-->nm.
4: singular_nounphrase-->singular_determiner, singular_noun.
5: plural_nounphrase-->plural_determiner, plural_noun.
6: plural_nounphrase-->plural_noun.
7: verbphrase-->singular_verb.
8: verbphrase-->plural_verb.
```

Fig. 14. The final solution for the `sentence` program

The resulted program will fail on the first two negative examples. The only negative example left which is covered by the program is: *"A dog jump"*. The clause that is found by the IDTS is $\texttt{sentence} \rightarrow \texttt{singular\_nounphrase}, \texttt{verbphrase}$. The set of its resolvents is: $\mathbf{C} = \{\texttt{sentence} \rightarrow \texttt{singular\_nounphrase}, \texttt{singular\_verb}$

and `sentence` $\rightarrow$ `singular_nounphrase, plural_verb`$\}$. The second clause in **C** is deleted because it does not occur in the refutation of positive examples. The final solution is listed in Figure 14.

The complete IMPUT session is presented in Appendix A. In this example we did not use the CPM testing facility. The printout in Appendix A presents the IMPUT session given the same positive and negative examples. The inferred program clauses are not transformed back to DCG because of lacking the DCG prettytyper module. On the other hand IMPUT can read DCG rules and then convert them into Prolog.

5 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* and the *shuttle*. The *shuttle* is described in [7].

We used the same testing technique as in [7] 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 have made ten tests on each domain computed the average accuracy and the average number of clauses in the resulting theory. The results are summarized in the following three tables.

The results of the benchmarks using: IMPUT algorithm with Prolog computation rule				
	Number of	Accuracy		
The domain	clauses	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				
	Number of	Accuracy		
The domain	clauses	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				
	Number of	Accuracy		
The domain	clauses	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%

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.

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.

6 The Extended PECG Application

In the following the extended PECG system [15] is presented which is currently the largest application of IMPUT. The extended PECG system integrates an ECG waveform classifier (called PECG) with IMPUT. The structure of the system can be seen in Figure 15

6.1 The PECG System

PECG [14] system classifies the incoming ECG waveforms. Normally it results in only few classes, therefore the diagnosis can be more easily established. All waveforms are put to the same class if they differ from each other in less extent. A similarity measure is used to solve this problem. A ten minute long ECG report may contain about 800 waveforms. To find several inappropriate among them requires much effort from the doctors.

The PECG system is based on an attribute grammar specification of ECG waveforms published by Skordalakis et al [32, 33]. The basic idea of PECG [14], is to integrate the ECG classifier program that has been implemented in Prolog with the IDTS [24] and a graphic viewer. This integrated tool can recognize whether any modification in the classifier program is needed. If the system cannot analyze the input then the user is helped by the built-in IDTS debugger to find the false clause.

In the PECG system the ECG waveforms are described by DCG [27]. The construction of these grammars for different waveforms is a very sophisticated task, therefore a tool which can assist in this work would be very useful. That was the reason for developing an *extended* version of the PECG system.

The user of this system has to prepare an *overly general* grammar in the first step. This grammar may accept not only correct but many incorrect waveforms as well. Of course it is much more easier to prepare such an *overly general* initial grammar than the correct one. The inputs of the extended PECG system are this initial grammar, some background knowledge, positive and negative examples of ECG waveforms to be inferred. The system uses IMPUT to improve the initial grammar such that the modified grammar accepts each positive example but no negative ones.

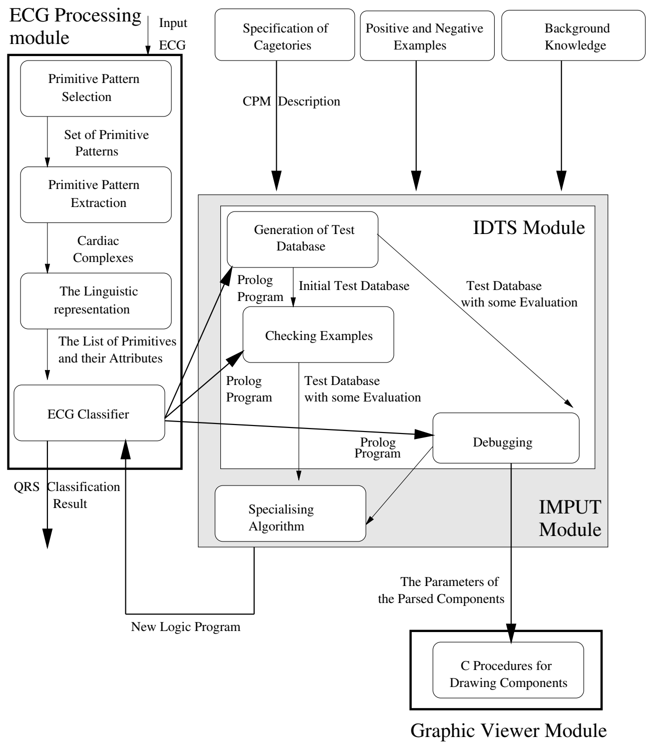

Fig. 15. The structure of the PECG system

During the learning process the system asks the user about the correctness of the recognized subpatterns (subgrammars). To answer these questions may be very difficult, hence the system has been augmented with a graphic viewer module, which displays the part of the ECG waveform being currently analyzed, see Figure 16. On the basis of the graphic display a medical expert can easily decide whether a subpattern of a given waveform has been correctly recognized or not. If an incorrect part of a grammar is identified then the system tries to apply an unfolding step to transform the grammar. This transformation usually results in a more precise recognizer of a given waveform. The system applies these transformation steps until the modified grammar does not recognize any negative example. By using this method correct recognizer grammars can be inferred for ECG waveforms starting from easy prepared overly general initial specifications.

6.2 The syntax of ECG grammar

In [33] an attribute grammar was used for the specification of ECG waveforms. An ECG waveform is described as a string of symbols from the alphabet $\Sigma = \{K^+, K^-, E, \Pi\}$

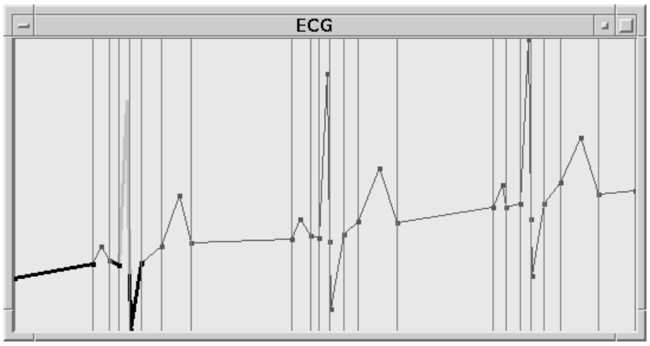

Fig. 16. The graphical representation of the recognized QRS part

where K^+ and K^- represent the positive and negative peaks, E the straight line segment and Π the parabolic segment.

Our approach starts from attribute grammar specifications of waveforms, however we use Definite Clause Grammar (DCG) representations for the recognition of these waveforms. The system was extended with a graphic viewer to trace the history of the traversing of the proof tree of the analyzer program.

6.3 PECG: A classifier program for ECG waveforms

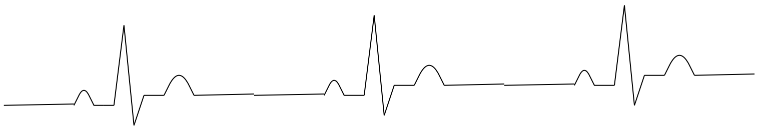

Fig. 17. An initial ECG waveform

As an example let us suppose that we have the following initial ECG waveform presented in Figure 17 The linguistic representation of it is the following:

$$EK^+\Pi K^+K^-EK^+\Pi K^+EK^+K^-\Pi K^+\Pi K^+\Pi K^+K^-EK^+\Pi$$

In Skordalakis' approach [33] many numeric values are associated with the primitives (e.g. boundaries of peaks, features of slopes, starting and ending points of segments). The PECG system uses these numeric features for the correct recognition of ECG waveforms. However, in this paper we concentrate only on the learning of the syntactical part, therefore these numeric features are not discussed here.

The *ECG classifier* module recognizes the ECG waveforms from their linguistic representations.

The IDTS part can be divided into three submodules. The *test database* module generates the initial test database from a CPM specification. In the *testing* part a predicate can be chosen and the user can give a representing element of the test frame generated for this predicate. Then he or she can decide whether, for the given input, the output is correct or not. The *debugging* part is based on the idea that if the program has already been tested then the test results can be directly applied without asking the user *difficult* questions.

In the PECG system the algorithmic debugger IDTS has been extended with a graphic viewer. This graphic module shows the ECG subpattern being analyzed and in this way provides effective assistance for the user to answer the questions invoked by the debugger.

6.4 The learning process

In the following we explain how the integrated system refines an initial DCG representation of a waveform. As an example a cardiac cycle description is selected from the whole ECG waveform description. Every cardiac complex contains one or more cardiac waves, but the most important phase in the heart action is the QRS complex. This part was chosen to demonstrate the learning process. In the original description a QRS complex can be built up from one to seven peaks and these clauses match the QRS pattern. We tried to refine this initial DCG program to recognize the correct series of peaks in the QRS complex.

The input description contains the program for classifying cardiac cycles, the background knowledge, the positive and negative examples (see Appendix B). The task of our integrated system was to find the correct program which can recognize only the positive examples.

The main issues of the session are the following. The user starts the program by entering `start` then enters the name of the file to be processed. All clauses read are numbered. The clauses obtained by unfolding are numbered by pairs of numbers `i-j` where the first number denotes the parent, the second denotes the substituted clause.

```
| ?- start.
Welcome to IMPUT learning system.
Please enter the filename to be processed: cardiac_cycle_desc.

 The background knowledge is:
17: t_or_p(A,B):-C(A,p_peak,C),C(C,n_peak,B)
...
24: segment(A,B):-C(A,par,B)

 The theory needs to be specialized is:
0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
1: cardiac_cycles-->B=A
2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
3: qrs-->peak(A,C),peaks(C,B)
```

```
...
16: p(A,B):-t_or_p(A,B)

 The positive examples are:
25: cardiac_cycles([p_peak,n_peak,line,...])
...
36: cardiac_cycles([n_peak,p_peak,n_peak,...])

 The negative examples are:
37: cardiac_cycles([p_peak,p_peak,line,...])
...
76: cardiac_cycles([n_peak,n_peak,p_peak,...])
```

The positive and negative examples are checked. At the beginning, both the positive and negative examples have to be covered by the initial description. Firstly, the system checks if the sets of positive and negative examples are distinct, secondly if there exist clauses which are not necessary to deduce the positive examples (in the case of the *most specific* version of IMPUT). If such clauses are exist then those will be removed from the program and the remaining clauses are listed. In this example clauses $7, 8, 9, 11, 12, 14, 15, 16$ are removed.

```
Checking input examples:

 The sets of positive and negative examples are distinct.

 Checking positive examples:

 Clauses were found, that are not needed to cover positive
  examples.
 - clauses were removed from the initial theory.

 The remained clauses are:
 0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
 1: cardiac_cycles-->B=A
 2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
 3: qrs-->peak(A,C),peaks(C,B)
 4: peaks-->peak(A,C),peaks(C,B)
 5: peaks-->B=A
 6: non_qrs-->sr(A,B)
10: sr-->segment(A,C),interwave_segment(C,B)
13: interwave_segment-->B=A
```

After that PECG checks for the negative examples. If the system finds a negative example covered by the initial program then the learning phase is invoked, otherwise the system has obtained the correct program. Only the covered negative examples are displayed.

The first step of the learning is to find the clause to be unfolded by means of the debugger. The questions asked by the system can be answered with *yes* or *no*. The answer *no* means that the debugger has found the false clause. In this case the debugging process is finished and the unfolding process can be started. In the opposite case the debugging goes on to the next predicate. In this example the last question is

```
 Is it ok [qrs([p_peak,p_peak])] (y/n) n
```

The answer is *no* because in the correct classification a positive peak can be followed only by negative peak.

```
37: cardiac_cycles([p_peak,p_peak,line,...[]) covered.
...
76: cardiac_cycles([n_peak,n_peak,p_peak,],[]) covered.

The fact cardiac_cycles([p_peak,p_peak,line,...],[])
 is covered by the theory.
Starting the false proc. algorithm to determine the basis
 of the unfolding.

 Is it ok [peaks([])] (y/n) y
 Is it ok [peaks([p_peak])] (y/n) y
 Is it ok [qrs([p_peak,p_peak])] (y/n) n],
```

The most complex part of the system is the unfolding part. As mentioned earlier, it is supposed that if a negative example is covered by the current version of the initial program, then there is at least one clause which is responsible for this incorrect covering. We have found that the clause $\mathtt{peaks} --> \mathtt{peak}(\mathtt{A},\mathtt{C}), \mathtt{peaks}(\mathtt{C},\mathtt{B})$ is incorrect. This clause cannot be removed from the initial program because the derivation of a positive example contains this clause.

```
Unfolding at the clause instance:
   3: qrs-->peak([p_peak, p_peak, line,...], [p_peak, line,...]),
            peaks([p_peak, line, p_peak,...], [line, p_peak,...])

 - trying resolvent(s): [3-1] actual minimum is: 12.5101.
 - trying resolvent(s): [3-2] actual minimum is: 0.

The result of the unfolding is:

0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
1: cardiac_cycles-->B=A
2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
3-4: qrs(A,B):-peak(A,C),peak(C,D),peaks(D,B)
3-5: qrs(A,B):-peak(A,C),B=C
...
```

The new program will not cover the first negative example, but covers the other ones. By repeatedly using the debugger we can find another erroneous clause. The last step of the learning process is the following:

```
Checking positive examples:

Checking negative examples:
```

```
66: cardiac_cycles([n_peak,p_peak,n_peak,...],[]) covered.

 The above theory:
    covers   12 positive samples from 12 (100.00%) and
    fails on 39 negative samples from 40 (97.50%).

 The fact cardiac_cycles([n_peak,p_peak,n_peak,...],[])
  is covered by the theory.
 Starting the false proc. algorithm to determine the basis
  of the unfolding.

 Unfolding at the clause instance:
 3-4-4-22-22-4-21-4-21-4-22-21-4:
 qrs([n_peak,p_peak,n_peak,...],[line,p_peak,par,...]):-
   C([n_peak,p_peak,n_peak,...],n_peak,[p_peak,n_peak,p_peak,...]),
   C([p_peak,n_peak,p_peak,...],p_peak,[n_peak,p_peak,n_peak,...]),
   C([n_peak,p_peak,n_peak,...],n_peak,[p_peak,n_peak,p_peak,...]),
   C([p_peak,n_peak,p_peak,...],p_peak,[n_peak,p_peak,p_peak,...]),
   C([n_peak,p_peak,p_peak,...],n_peak,[p_peak,p_peak,line,]),
   C([p_peak,p_peak,line,...],p_peak,[p_peak,line,p_peak,...]),
   peak([p_peak,line,p_peak,...],[line,p_peak,par,...]),
   peaks([line,p_peak,par,...][line,p_peak,par,...])

 - trying resolvent(s): [3-4-4-22-22-4-21-4-21-4-22-21-4-7]
   actual minimum is: 3.44203.
 - trying resolvent(s): [3-4-4-22-22-4-21-4-21-4-22-21-4-8]
   actual minimum is: 3.52546.

 The result of the unfolding is:

0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
1: cardiac_cycles-->B=A
2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
3-4-4-21-21-4-4-22-4-22-4-21: qrs(A,B):-
  C(A,p_peak,C),C(C,n_peak,D),C(D,p_peak,E),C(E,n_peak,F),
  C(F,p_peak,G),C(G,n_peak,H),C(H,p_peak,I),peaks(I,B)
...

 Checking positive examples:

 Checking negative examples:

 The above theory:
    covers   12 positive samples from 12 (100.00%) and
    fails on 40 negative samples from 40 (100.00%).
```

The learning process is finished because the program above has covered all positive examples and failed on all negative ones. The complete result can be found in Appendix B

7 Related Work

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

In contrast to the specialization techniques in [17, 4], 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 [31, 5, 23, 26, 9, 20, 30, 35, 10, 25, 36]. 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 [35] is clause removal. In contrast to SPECTRE, the only clauses that are considered for removal in this approach 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 [36] 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 [31, 5, 29, 26, 30, 25], 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 [29]). The branching factor in the space of possible refinements is normally much larger for these approaches than for SPECTRE.

The major difference between SPECTRE and previous techniques that use goal reduction for specializing logic programs (ML-SMART [5], ANA-EBL [9], FOCL [26], GRENDEL [10], FOCL-FRONTIER [25]) is the way in which the search for a specialization is performed. The algorithms ANA-EBL, FOCL, GRENDEL and FOCL-FRONTIER, like MIS [31] 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 [19]), while SPECTRE searches the refinement graph once (cf. ID3 [28]). 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 [7] 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 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 [6] 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.

8 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 [11].

The IMPUT system which integrates SPECTRE and IDTS methods has been fully implemented, but there are still examples that IMPUT can not solve:

1. ```
   p(X).

   positive p(f(a)).
   negative p(g(a)).
   ```
2. ```
   p(X).

   positive p((a, b)).
   negative p((a, b, c)).
   ```
3. ```
   p(X) :- is_list(X).
   ```
   ```

```
positive p([a]).
negative p([(a,b)]).
```

The reason of that IMPUT can not solve the above three examples is that the IMPUT algorithm can not do any subsumption on the variables of clauses. When IMPUT finds a unit-clause to be unfolded it will fail because it finds no literals on the right side of the corresponding clause. It will return saying "there is no possible unfolding any more". At that point the algorithm terminates. Nienhuys-Cheng and Wolf described this situation in [21] and gave a theoretically complete learning algorithm. To incorporate subsumption in IMPUT is 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. Alexin, Z., Gyimóthy, T., Boström, H.: "Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner" *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI96)* Budapest, Hungary, John Wiley & Sons, Ltd. (1996) 403–407
4. Bain, M., Muggleton, S.: "Non-Monotonic Learning", in Muggleton S. (ed.), *Inductive Logic Programming*, Academic Press, London (1992) 145–161
5. 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
6. Boström, H.: "Specialization of Recursive Predicates", *Proceedings of the Eighth European Conference on Machine Learning*, Springer-Verlag (1995) 92–106
7. 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
8. Boye, J., Paakki, J., Maluszyński, J.: "Synthesis of Directionality Information for Functional Logic Programs" in Proc. 3rd International Workshop on Static Analysis, Padova, 1993. LNCS 724, Springer-Verlag (1993) 165–177
9. Cohen, W. W.: "The Generality of Overgenerality", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 490–494
10. Cohen, W. W.: "Compiling Prior Knowledge Into an Explicit Bias", *Machine Learning: Proceedings of the Ninth International Workshop*, Morgan Kaufmann (1992) 102–110
11. De Raedt, L., Bruynooghe, M.: "Belief Updating from Integrity Constraints and Queries" *Artificial Intelligence*, 53, 2–3, (1992) 291–307
12. Gyimóthy, T., Paakki, J.: "Static Slicing of Logic Programs" in Proc. Automated and Algorihmic Debugging ed. Ducassé, M. Sain-Malo France (1995) 65–77
13. 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

14. Kókai, G., Alexin, Z., Gyimóthy, T.: Classifying ECG Waveforms in Prolog Proc. of the Fourth International Conference on The Practical Application of PROLOG (PAP96) London, United Kingdom April 23-25, (1996) 193–221
15. Kókai, G., Alexin, Z., Gyimóthy, T.: "Application of Inductive Logic Programming for Learning ECG Waveforms" accepted to the Artificial Intelligence in Medicine (AIME97) Conference Grenoble 1997. as a poster
16. Knuth, D. E.: "Semantics of context-free languages" *Mathematical Systems Theory* **2** (1968) 127–145
17. Ling, C. X.: "Non-Monotonic Specialization", *Proceedings of International Workshop on Inductive Logic Programming*, Portugal (1991) 59–68
18. Lloyd, J. W.: Foundations of Logic Programming, (2nd edition) Springer-Verlag (1987)
19. Michalski, R. S.: "Pattern Recognition as Rule-Guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, bf 2 (1980) 349–361
20. Mooney, R. J., Ourston, D.: "Constructive Induction in Theory Refinement", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 178–182
21. Nienhuys-Cheng, S.-H., de Wolf, R.: "A Complete Method for Program Specialization Based on Unfolding" *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI96)* Budapest, Hungary, John Wiley & Sons, Ltd. (1996) 438–442
22. Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM **31:6** June (1988) 676–686
23. 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
24. 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
25. 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
26. 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
27. Pereira, F. C. N., Warren, D. H. D: "Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks" *Artificial Intelligence*, **13** (1980) 231–278
28. Quinlan, J. R.: "Induction of Decision Trees", *Machine Learning* **1** (1986) 81–106
29. Quinlan, J. R.: "Learning Logical Definitions from Relations", *Machine Learning* **5** (1990) 239–266
30. Richards, B. L., Mooney, R. J.: "First-Order Theory Revision", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 447–451
31. Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)
32. Skordalakis, E., Papakonstantinou G.: Towards an Attribute Grammar for the Description of ECG Waveforms 7th International Conference on Pattern Recognition (1984).
33. Skordalakis, E.: ECG Analysis in Syntactic and Structural Pattern Recognition Theory and Applications ed. Bunke, H. and Sanfeliu, A. World Scientific (1990) 499–533
34. 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
35. Wogulis, J.: "Revising Relational Domain Theories", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 462–466

36. 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

Appendix A - A complete learning session by IMPUT

```
| ?- start.
Welcome to IMPUT learning system.
please enter the filename to be processed: sentence.

 The background knowledge is:

9: nm([bob|A],A)
10: singular_determiner([a|A],A)
11: singular_determiner([the|A],A)
12: plural_determiner([the|A],A)
13: singular_noun([boy|A],A)
14: singular_noun([dog|A],A)
15: plural_noun([boys|A],A)
16: plural_noun([dogs|A],A)
17: singular_verb([runs|A],A)
18: singular_verb([plays|A],A)
19: singular_verb([jumps|A],A)
20: plural_verb([run|A],A)
21: plural_verb([play|A],A)
22: plural_verb([jump|A],A)

 The theory needs to be specialized is:

0: sentence(A,B):-nounphrase(A,C),verbphrase(C,B)
1: nounphrase(A,B):-singular_nounphrase(A,B)
2: nounphrase(A,B):-plural_nounphrase(A,B)
3: singular_nounphrase(A,B):-nm(A,B)
4: singular_nounphrase(A,B):-singular_determiner(A,C),
                             singular_noun(C,B)
5: plural_nounphrase(A,B):-plural_determiner(A,C),
                           plural_noun(C,B)
6: plural_nounphrase(A,B):-plural_noun(A,B)
7: verbphrase(A,B):-singular_verb(A,B)
8: verbphrase(A,B):-plural_verb(A,B)

 The positive examples are:

23: sentence([bob,plays],[])
24: sentence([dogs,run],[])
25: sentence([the,boy,jumps],[])
26: nounphrase([bob],[])
27: nounphrase([the,dogs],[])
28: verbphrase([run],[])
```

```
29: verbphrase([runs],[])

 The negative examples are:

30: sentence([the,boys,plays],[])
31: sentence([boys,runs],[])
32: sentence([a,dog,jump],[])

Checking input examples:

 The sets of positive and negative examples are distinct.

 Checking positive examples:

 Checking negative examples:

30: sentence([the,boys,plays],[]) covered.
31: sentence([boys,runs],[]) covered.
32: sentence([a,dog,jump],[]) covered.

 The fact sentence([the,boys,plays],[]) is covered by the theory.
 Starting the false proc. algorithm to determine
          the basis of the unfolding.

 Is it ok [plural_nounphrase([the,boys,plays],[plays])] (y/n) y
 Is it ok [nounphrase([the,boys,plays],[plays])] (y/n) y
 Is it ok [verbphrase([plays],[])] (y/n) y
 Is it ok [sentence([the,boys,plays],[])] (y/n) n

 Unfolding at the clause instance:
   0: sentence([the,boys,plays],[]):-
      nounphrase([the,boys,plays],[plays]),
      verbphrase([plays],[])

 - trying resolvent(s): [0-1]
          actual minimum is: 10.18355343004271.
 - trying resolvent(s): [0-2]
           actual minimum is: 10.390359525563188.

 The result of the unfolding is:

0-1: sentence(A,B):-singular_nounphrase(A,C),
     verbphrase(C,B)
0-2: sentence(A,B):-plural_nounphrase(A,C),
     verbphrase(C,B)
1: nounphrase(A,B):-singular_nounphrase(A,B)
2: nounphrase(A,B):-plural_nounphrase(A,B)
3: singular_nounphrase(A,B):-nm(A,B)
4: singular_nounphrase(A,B):-singular_determiner(A,C),
   singular_noun(C,B)
```

```
5: plural_nounphrase(A,B):-plural_determiner(A,C),
   plural_noun(C,B)
6: plural_nounphrase(A,B):-plural_noun(A,B)
7: verbphrase(A,B):-singular_verb(A,B)
8: verbphrase(A,B):-plural_verb(A,B)

 Checking positive examples:


 Checking negative examples:

30: sentence([the,boys,plays],[]) covered.
31: sentence([boys,runs],[]) covered.
32: sentence([a,dog,jump],[]) covered.

 The above theory:
    cover   7 positive samples from 7 (100.00%) and
    fail on 0 negative samples from 3 (0.00%).

 The fact sentence([the,boys,plays],[]) is covered by the theory.
 Starting the false proc. algorithm to determine
          the basis of the unfolding.

 Unfolding at the clause instance:
   0-2: sentence([the,boys,plays],[]):-
        plural_nounphrase([the,boys,plays], [plays]),
        verbphrase([plays],[])

 - trying resolvent(s): [0-2-1]
          actual minimum is: 13.368065554250645.
 - trying resolvent(s): [0-2-2]
          actual minimum is: 12.613178052087179.

 The result of the unfolding is:

0-1: sentence(A,B):-singular_nounphrase(A,C),
     verbphrase(C,B)
0-2-8: sentence(A,B):-plural_nounphrase(A,C),
       plural_verb(C,B)
1: nounphrase(A,B):-singular_nounphrase(A,B)
2: nounphrase(A,B):-plural_nounphrase(A,B)
3: singular_nounphrase(A,B):-nm(A,B)
4: singular_nounphrase(A,B):-singular_determiner(A,C),
   singular_noun(C,B)
5: plural_nounphrase(A,B):-plural_determiner(A,C),
   plural_noun(C,B)
6: plural_nounphrase(A,B):-plural_noun(A,B)
7: verbphrase(A,B):-singular_verb(A,B)
8: verbphrase(A,B):-plural_verb(A,B)
```

```
 Checking positive examples:


 Checking negative examples:

32: sentence([a,dog,jump],[]) covered.

 The above theory:
    cover   7 positive samples from 7 (100.00%) and
    fail on 2 negative samples from 3 (66.67%).

 The fact sentence([a,dog,jump],[]) is covered by the theory.
 Starting the false proc. algorithm to determine
          the basis of the unfolding.

 Is it ok [singular_nounphrase([a,dog,jump],[jump])] (y/n) y
 Is it ok [verbphrase([jump],[])] (y/n) y
 Is it ok [sentence([a,dog,jump],[])] (y/n) n

 Unfolding at the clause instance:
   0-1: sentence([a,dog,jump],[]):-
        singular_nounphrase([a,dog,jump], [jump]),
        verbphrase([jump],[])

 - trying resolvent(s): [0-1-1]
          actual minimum is: 4.141709450076293.
 - trying resolvent(s): [0-1-2]
          actual minimum is: 3.900134529890125.

 The result of the unfolding is:

0-1-7: sentence(A,B):-singular_nounphrase(A,C),
       singular_verb(C,B)
0-2-8: sentence(A,B):-plural_nounphrase(A,C),
       plural_verb(C,B)
1: nounphrase(A,B):-singular_nounphrase(A,B)
2: nounphrase(A,B):-plural_nounphrase(A,B)
3: singular_nounphrase(A,B):-nm(A,B)
4: singular_nounphrase(A,B):-singular_determiner(A,C),
   singular_noun(C,B)
5: plural_nounphrase(A,B):-plural_determiner(A,C),
   plural_noun(C,B)
6: plural_nounphrase(A,B):-plural_noun(A,B)
7: verbphrase(A,B):-singular_verb(A,B)
8: verbphrase(A,B):-plural_verb(A,B)

 Checking positive examples:


 Checking negative examples:
```

```
 The above theory:
    cover   7 positive samples from 7 (100.00%) and
    fail on 3 negative samples from 3 (100.00%).

 ---------------- The final result theory is: ----------------

0-1-7: sentence(A,B):-singular_nounphrase(A,C),
       singular_verb(C,B)
0-2-8: sentence(A,B):-plural_nounphrase(A,C),
       plural_verb(C,B)
1: nounphrase(A,B):-singular_nounphrase(A,B)
2: nounphrase(A,B):-plural_nounphrase(A,B)
3: singular_nounphrase(A,B):-nm(A,B)
4: singular_nounphrase(A,B):-singular_determiner(A,C),
   singular_noun(C,B)
5: plural_nounphrase(A,B):-plural_determiner(A,C),
   plural_noun(C,B)
6: plural_nounphrase(A,B):-plural_noun(A,B)
7: verbphrase(A,B):-singular_verb(A,B)
8: verbphrase(A,B):-plural_verb(A,B)
```

Appendix B - An example session using PECG

The input for the extended PECG

```
0: cardiac_cycles --> cardiac_cycle, cardiac_cycles.
1: cardiac_cycles --> {true}.

2: cardiac_cycle  --> qrs, non_qrs.

3: qrs     --> peak,peaks.
4: peaks   --> peak,peaks.
5: peaks   --> {true}.

6: non_qrs --> sr.
7: non_qrs --> interwave_segment, t, interwave_segment .
8: non_qrs --> interwave_segment, p, interwave_segment .
9: non_qrs --> interwave_segment, t, interwave_segment, p,
               interwave_segment.

10: sr  --> segment, interwave_segment.
11: sr  --> peak, interwave_segment.

12: interwave_segment --> segment, interwave_segment.
13: interwave_segment --> {true}.
14: interwave_segment --> peak, interwave_segment .
```

```
15: t  --> t_or_p.
16: p  --> t_or_p.

17: background (t_or_p  --> [p_peak], [n_peak]).
18: background (t_or_p  --> [n_peak], [p_peak]).
19: background (t_or_p  --> [p_peak]).
20: background (t_or_p  --> [n_peak]).
21: background (peak    --> [p_peak]).
22: background (peak    --> [n_peak]).
23: background (segment --> [line]).
24: background (segment --> [par]).

25: positive cardiac_cycles([p_peak,n_peak,line,...],[]).
26: positive cardiac_cycles([n_peak,p_peak,line,...],[]).
27: positive cardiac_cycles([n_peak,p_peak,n_peak,...],[]).
28: positive cardiac_cycles([p_peak,n_peak,p_peak,...],[]).
...
37: negative cardiac_cycles([p_peak,p_peak,line,...],[]).
38: negative cardiac_cycles([n_peak,n_peak,line,p_peak,...],[]).
39: negative cardiac_cycles([p_peak,p_peak,n_peak,...],[]).
40: negative cardiac_cycles([n_peak,n_peak,p_peak,...],[]).
...
```

The output of the extended PECG

```
0: cardiac_cycles --> cardiac_cycle,cardiac_cycles
1: cardiac_cycles --> {true}
2: cardiac_cycle  --> qrs,non_qrs
3-4-4-21-21-4-4-22-4-22-4-21: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak],
  [n_peak],[p_peak],peaks

3-4-4-21-21-4-4-22-4-22-5: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak],[n_peak]

3-4-4-21-21-4-4-22-5-21: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak]

3-4-4-21-21-4-5-22: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak]

3-4-4-21-21-5: qrs --> [p_peak],[n_peak],[p_peak]

3-4-4-22-22-4-21-4-21-4-22-21-4-22: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak],
  [p_peak],[n_peak],peaks

3-4-4-22-22-4-21-4-21-4-22-21-5: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak],[p_peak]
```

```
3-4-4-22-22-4-21-4-21-5-22: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak]

3-4-4-22-22-4-21-5: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak]

3-4-4-22-22-5: qrs --> [n_peak],peak,[n_peak]
3-4-5-21-22: qrs --> [n_peak],[p_peak]
3-4-5-22-21: qrs --> [p_peak],[n_peak]
3-5: qrs --> peak
5: peaks --> {true}
6: non_qrs --> sr
10: sr --> segment,interwave_segment
13: interwave_segment --> {true}
```