Analyzing and Learning ECG Waveforms

Gabriella Kókai[1], Zoltán Alexin[2], and Tibor Gyimóthy[3]

[1] Institute of Informatics, József Attila University
Árpád tér 2, H-6720 Szeged, Hungary
Phone: (36) +(62) 311184, Fax: (36) +(62) 312292
e-mail: kokai@inf.u-szeged.hu

now visiting the Chair of Programming Languages,
Department of Computer Science,
Friedrich-Alexander University of Erlangen-Nürnberg
Martenstr. 3. D-91058 Erlangen, Germany
e-mail: kokai@informatik.uni-erlangen.de

[2] Department of Applied Informatics, József Attila University
Árpád tér 2, H-6720 Szeged, Hungary
Phone: (36) +(62) 311184, Fax: (36) +(62) 312292
e-mail: alexin@inf.u-szeged.hu

[3] Research Group on Artificial Intelligence
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.

Abstract. In this paper we present a system which integrates an ECG waveform classifier (called PECG) with an interactive learner (called IMPUT). The PECG system is based on an attribute grammar specification of ECGs that has been transformed to Prolog. The IMPUT system combines the interactive debugging technique IDT with the unfolding algorithm introduced in SPECTRE. The main result achieved in the new version of the PECG system is that an ILP method can be used to improve the effectiveness of a real size Prolog application. Applying the IMPUT method, the extended PECG system is able to suggest a correct solution to the user to replace the buggy clause recognized during the debugging process. [4]

1 Introduction

In this paper we present a system which integrates an ECG waveform classifier (called PECG) with an interactive learner (called IMPUT).

The PECG system contains three main parts: the ECG processing, the IDT and graphic viewer modules. It is based on an attribute grammar approach of ECG waveform analysis published by Skordalakis et al [16, 17].

[4] This work was supported by the grant OTKA T14228 and the project PHARE TDQM 9305-02/1022 ("ILP2/HUN").

Skordalakis's system can classify the recognized ECG waveforms according to certain synthesized attributes. It contains a preprocessing part which produces a linguistic representation from an ECG signal. This linguistic representation is used as the input for the PECG system. The basic idea of PECG, introduced in [8], is to integrate the ECG classifier program (implemented in Prolog) with the IDT algorithmic debugger module [7, 13] and a graphic viewer. This integrated tool can recognize if any modification in the classifier program is needed. If the system cannot analyze the input then the user is helped by the built-in IDT (Interactive Diagnosis and Testing) debugger to find the false clause. The IDT improves the Shapiro's original debugging method [15] and can be employed in the debugging and testing of Prolog programs [7, 13]. The graphic viewer module helps the users to decide whether the program with a given input works correctly or not, by displaying the steps of debugging phase.

The learning part of the PECG based on an Inductive Logic Programming (ILP) [9, 11] method called IMPUT introduced in [1]. The IMPUT system combines the interactive debugging technique IDT with the specialization algorithm SPECTRE presented in [2]. The main points behind the IMPUT method 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 IDT is used to identify a buggy clause instance. This clause is removed from the initial program. If a derivation of a positive example contains this clause, then those resolvents of the clause which appeared in this derivation are added to the initial program. A modified version of the impurity measure strategy is used to determine the literal to be unfolded [1]. The former version of the IMPUT system was modified such that it can analyze a Prolog program written in DCG form containing recursive predicates.

The new method furnished in this paper involves an integration of PECG with the IMPUT interactive learning system. With the help of this system not only the place of the error can be located in the program but in many cases starting from a *buggy* Prolog program we can infer a *correct* program using an unfolding transformation and the IDT debugger. Applying the IMPUT method, the extended PECG system is able to suggest a correct solution to the user to replace the buggy clause recognized during the debugging process.

The main result achieved in the new version of the PECG system presented in this paper is that an ILP method can be used to improve the effectiveness of a real size Prolog application. The work of this integrated method is presented in an example in Section 4. Using the IMPUT system we can effectively assist in preparing the correct description of the basic structures of ECG waveforms. However, we found that this method is not really suitable for the learning rules which describe the computation of the values of the arguments of these structures. Though unproven as yet, it is supposed that other ILP systems (Progol [10], LINUS [5], CLAUDINE [3]) will be aptly suited to this task.

Sections 2 and 3 contain, respectively short descriptions of the PECG and

IMPUT systems. In Section 4 the workings of the integrated system is demonstrated. Finally, in Section 5 a brief summary and comments on possible future studies are presented.

2 The PECG system

In this section we give a short overview of the PECG system. A more detailed description can be found in [8].
The structure of the system can be seen in Figure 1.

The main parts are the following:

- *the ECG processing module:* this part of the system uses a syntactic approach to recognize an electrocardiographic pattern.
- *the IDT module:* this part is an interactive debugging and testing tool that is based on the Shapiro's Interactive Diagnosis Algorithm [15] and Category Partition Testing Method [12].
- *the graphic viewer module:* this part makes visible the current state of the debugging process.

The ECG processing module is built up from the following modules:

- *primitive pattern selection*
- *primitive pattern extraction*
- *the linguistic representation*
- *the ECG classifier*

In the *primitive pattern selection* part an ECG waveform can be considered as a composite pattern which can be decomposed as soon as possible into primitive patterns (straight line, peak, parabola). The main task of the *primitive pattern extraction* is to select the peaks. This part is based on recognition of the noisy peaks. The real peaks are found from the difference between the set of all peaks and the noisy ones. In the *linguistic representation* part the result of the last two steps are represented as a string of symbols from an alphabet. The *ECG classifier* module recognizes the ECG waveforms from their linguistic representations.

The IDT 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 testframe 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 IDT has been extended by adding a graphic viewer. This graphic module shows the ECG waveform being analyzed and in this way provides effective assistance for the user to answer the questions invoked by the debugger. At the beginning of the debugging phase the whole ECG waveform is displayed (see Figure 2) and in each step of the debugging process the current component of the input is presented.

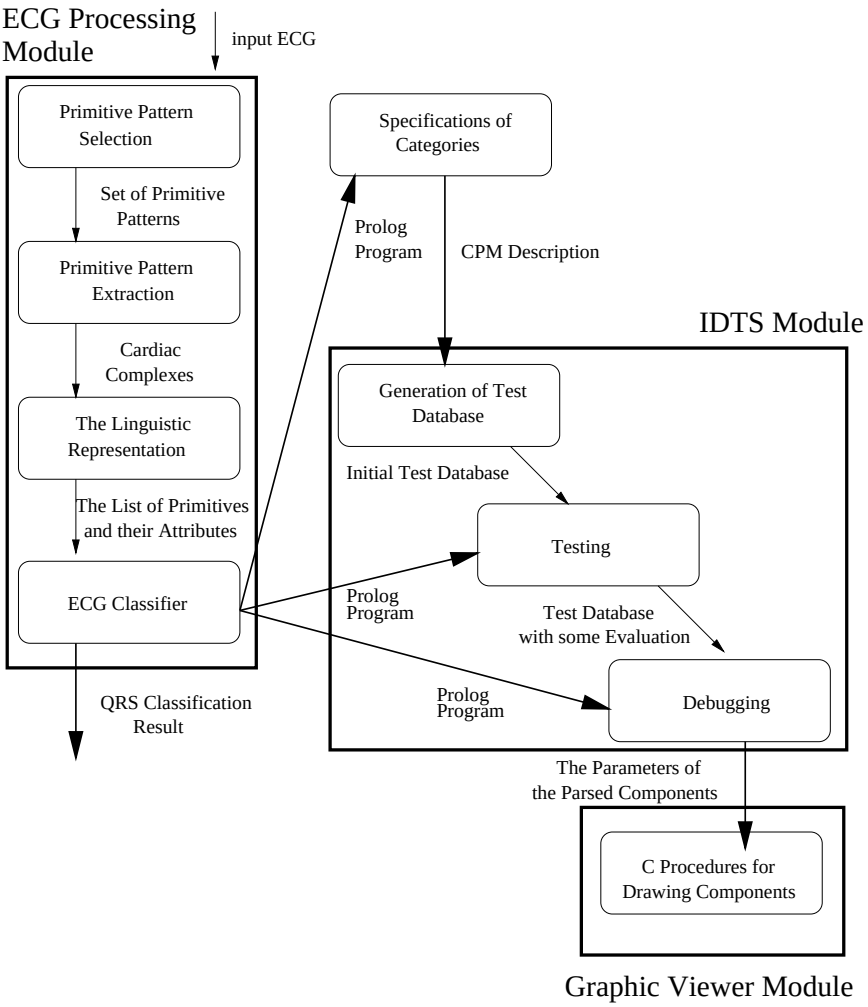

Fig. 1. The structure of the PECG system

In the Figure 3 it can be seen, that the graphic viewer draws out the whole recognized QRS waveform in a different graphic form. This module contains C routines.

3 The IMPUT system

In this section the IMPUT system is briefly presented. It consists of two main parts. The specialization algorithm comes from the SPECTRE system while the interactive debugger part is imported from the IDT. The original specialization

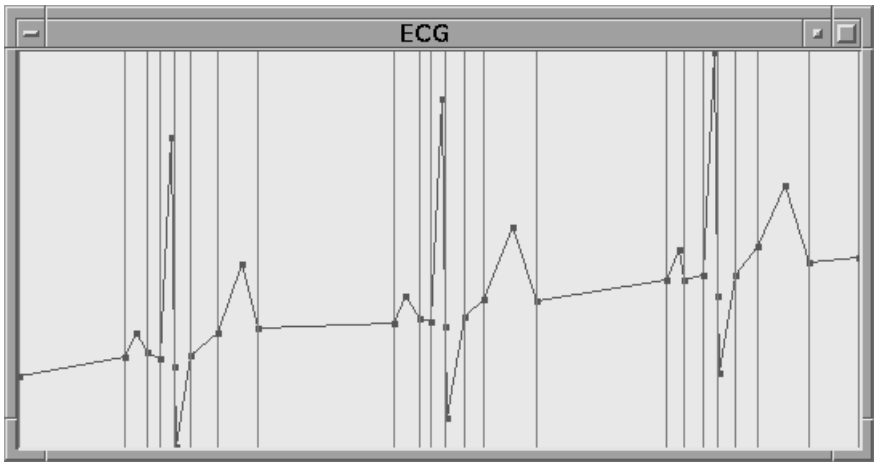

Fig. 2. The representation of the input ECG

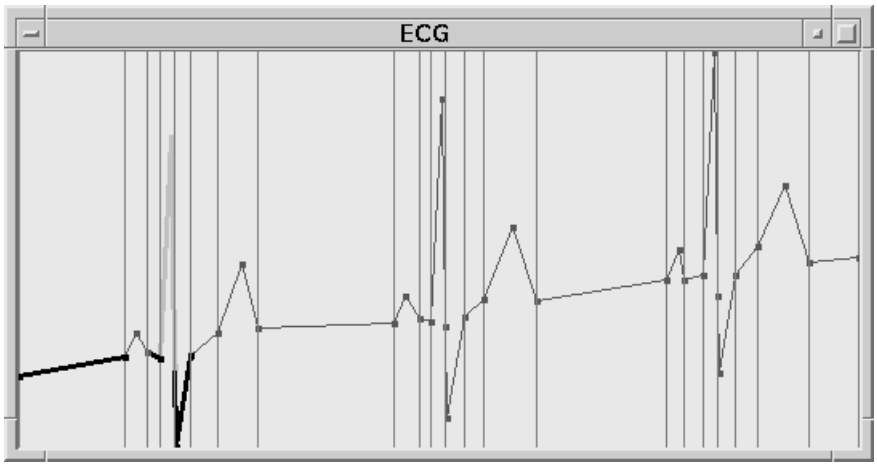

Fig. 3. The graphical representation of the recognized QRS part

algorithm was extended in order to be able to revise multiple predicates simultaneously, to be able to read in DCG rules and to handle recursive predicates. This modification initiated some further improvements and these are incorporated in IMPUT as well.

The algorithm SPECTRE [2] specializes logic programs with respect to positive and negative examples by applying the transformation rule *unfolding* [18] together with clause removal. The choice of which literal to unfold upon is made

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 $\mathbf{P}'^{(i+1)} = \tilde{\mathcal{U}}\left(\mathbf{P}'^{(i)}\right)$ $(0 \leq i < n)$, $\tilde{\mathcal{U}}$ is the unfolding operator extended with clause removal.

The Algorithm:

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

Fig. 4. The basic algorithm of IMPUT

using a computation rule, which is given as input to the algorithm. A detailed description of SPECTRE can be found in [2].

The IDT method is based on the algorithmic program debugging technique introduced by Shapiro [15]. This method can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly. A major drawback of this debugging technique is the great number of queries made to the user about the correctness of intermediate results of procedure calls. An improvement in the bug localization process is realized in IDT by combining the *category partition testing method* [6], with the algorithm introduced in [15]. The main idea basically as follows: 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 IDT interactive debugging environment was presented in [7, 13] and a similar system for imperative languages was introduced in [6].

The IMPUT system integrates the SPECTRE and the IDT methods. 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 that when a negative example is covered by the current version of the program, then there is at least one clause which is responsible for this incorrect covering. IMPUT uses the IDT 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.

The basic algorithm of IMPUT system is presented in Figure 4. A more detailed descriptions of the system can be found in [1].

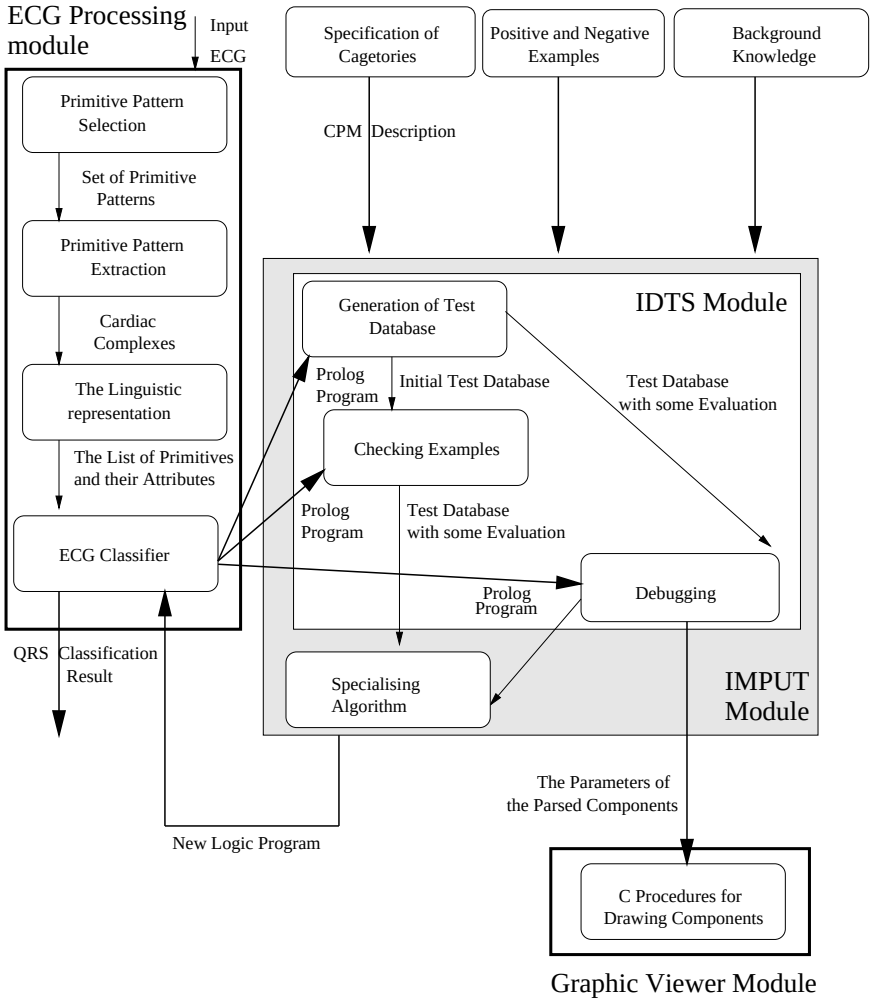

Fig. 5. The structure of the extended PECG system

4 Integration of PECG and IMPUT

In the following the extended PECG system is presented. We explain how the integrated system can learn an ECG description. The structure of the extended PECG system is shown in Figure 5. As mentioned earlier, the debugger is extended with a graphic viewer to trace the history of the traversing of the proof tree of the ECG analyzer program, but to keep our example as simple as possible here we do not present the results any of these displays.

4.1 The syntax of ECG grammar

In Skordalakis's paper [17] an attribute grammar was presented for recognition of ECG waveforms. A part of this description is selected to demonstrate the learning of one cardiac cycle of a normal ECG. The syntactic part of the grammar in Prolog representation is presented in Appendix 6.1.

The syntax of the attribute grammar which we use here is basically the one by Udupa [19] for the normal ECGs. The ECG waveform is encoded as a string of pairs. The first element of each pair is a slope symbol and the second element is a number representing the duration of the corresponding line subsegment.

The terminal symbols of the grammar are the following:

l denoting line segment of unit length and large positive slope,
unit length = 0.25 mm and slope = (75° , 90°)
nl denoting line segment of unit length and large negative slope,
unit length = 0.25 mm and slope = (−75° ,−90°)
i denoting line segment of unit length and intermediate positive slope,
unit length = 0.25 mm and slope = (35°, 75°)
ni denoting line segment of unit length and intermediate negative slope,
unit length = 0.25 mm and slope = (−35°, −75°)
s denoting line segment of unit length and small positive slope
unit length = 0.25 mm and slope = (5° , 35°)
ns denoting line segment of unit length and small negative slope,
unit length = 0.25 mm and slope = (5° , 35°)
h denoting horizontal line segment e.g. slope = (−5° , 5°)

The nonterminal symbols to be used are the following:

cardiac_cycle	denoting a cardiac cycle
Q, RS, T, P	denoting a Q, RS, T, P wave, respectively
ST, PR, TP	denoting a ST, PR, TP segment, respectively
C, D, E, F, G, H, I, K, L, M	denoting auxiliary nonterminals

Our approach is based on the attribute grammar given in [17] but we use a logical program representation for the recognition of the cardiac cycle. The attribute grammar presented in [19] was used as a starting-point and was rewritten in Prolog. The difference between an attribute grammar representation and a

Prolog program, however made the rewriting process non-trivial. We used definite clause grammar (DCG) [14] representation for that [4].

4.2 The learning process

In the following we present how the IMPUT learning system refines the initial DCG program. To demonstrate the work of IMPUT some changes were made in the ECG gammar.

- A new version of the nonterminal *l* was added to the description;

```
l --> negative_intermediate_slope.
l --> negative_large_slope.
l --> negative_small_slope.
```

- A new nonterminal *n* was introduced;

```
n --> large_slope.
n --> intermediate_slope.
```

- The *large_slope* nonterminal was replaced by *n* in the following DCG rules;

```
e --> n, e.
e --> {true}.
c --> n, c.
c --> n.
```

- The *negative_large_slope* nonterminal was changed to *l* in the following DCG rules.

```
d --> l, d.
d --> l.
```

The input description contains the program for classifying cardiac cycles, the background knowledge, the positive and negative examples (see Appendix 6.2). 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: ecg_description.

 The background knowledge is:
38: negative_large_slope(A,B):-C(A,nl,B)
...
44: horizontal_slope(A,B):-C(A,h,B)
```

```
 The theory needs to be specialized is:
0: main(A,[]):-cardiac_cycle(A,[])
1: cardiac_cycle(A,B):-q(A,C),rs(C,D),st(D,E),t(E,F),
                       tp(F,G),p(G,H),pr(H,B)
2: q(A,B):-l(A,C),q(C,B)
3: q(A,B):-B=A
...

36: pr(A,B):-i(A,C),pr(C,B)
37: pr(A,B):-B=A

 The positive examples are:
45: main([ni,ni,l,l,nl,...],[])
...
51: main([ni,nl,l,nl,nl,...],[])

 The negative examples are:
52: main([ns,ni,ni,l,l,...],[])

57: main([nl,ni,l,l,nl,...],[])
```

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. If such clauses are exist then those will be removed from the program. In this case the program found that the following clause

```
6: l(A,B):-negative_small_slope(A,B)
```

is superfluous in the description. It means that the extended PECG system recognized the first error which we made. The superfluous clause was left out and the remaining clauses were listed.

```
 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: main(A,[]):-cardiac_cycle(A,[])
1: cardiac_cycle(A,B):-q(A,C),rs(C,D),st(D,E),t(E,F),
                       tp(F,G),p(G,H),pr(H,B)
2: q(A,B):-l(A,C),q(C,B)
3: q(A,B):-B=A
4: l(A,B):-negative_intermediate_slope(A,B)
5: l(A,B):-negative_large_slope(A,B)
```

```
7: rs(A,B):-c(A,C),d(C,D),e(D,B)
....
```

After this PECG checks for the negative examples. If the system finds such a negative example that is covered by the initial program then the learning phase is begun, otherwise the system will stop. Only the covered negative examples are displayed.

The first step of the learning is to find the basis of the unfolding by 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, 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 [c([i,l,nl,l,ns,i,h,ns,ns,ns,ni,ns,ns,i,h,ns,ns,ns,ni,ns,s],
  [nl,l,ns,i,h,ns,ns,ns,ni,ns,ns,i,h,ns,ns,ns,ni,ns,s])] (y/n) n
```

The answer is *no* because in the correct classification the c can contain only *large_slope* values. It is the second change made that allows the deduction of *intermediate_slope* from c too.

```
 Checking negative examples:
54: main([ni,nl,ni,nl,l,...],[])  covered.
55: main([nl,nl,l,l,i,...],[])    covered.
56: main([ni,nl,l,l,nl,...],[])   covered.
57: main([nl,ni,l,l,nl,...],[])   covered.

 The fact main([ni,nl,ni,nl,l,...],[]) is covered by the theory.
 Starting the false proc. algorithm to determine the basis of the unfolding.

 Is it ok [ l([ni,nl,ni,nl,l,...],[nl,ni,nl,l,i,...]) ] (y/n) y
 Is it ok [ l([nl,ni,nl,l,i,],[ni,nl,l,i,l,...]) ] (y/n) y
...
 Is it ok [ c([i,l,nl,l,ns,...],[nl,l,ns,i,h,...]) ] (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 $\mathtt{c(A,B)} :- \mathtt{n(A,C)}, \mathtt{c(C,B)}$ is incorrect. This clause cannot be removed from the initial program because the derivation of a positive example contains this clause. The resolvents of this clause are: $\mathtt{c(A,B)} :- \mathtt{large_slope(A,C)}, \mathtt{c(C,B)}$ and $\mathtt{c(A,B)} :- \mathtt{intermediate_slope(A,C)}$, $\mathtt{c(C,B)}$ The second clause $\mathtt{c(A,B)} :- \mathtt{intermediate_slope(A,C)}, \mathtt{c(C,B)}$ is removed, because there is no positive example whose derivation contains it. The system displays the clause to be unfolded and the result of the unfolding.

```
Unfolding at the clause instance:
   8: c([i,l,nl,l,ns,...],[nl,l,ns,i,h,...]):-
          n([i,l,nl,l,ns,...],[l,nl,l,ns,i,...]),
          c([l,nl,l,ns,i,...],[nl,l,ns,i,h,...])
```

```
 - trying resolvent(s): [8-1] actual minimum is:  6.877840558577583.
 - trying resolvent(s): [8-2] actual minimum is: 11.344977967946406.

 The result of the unfolding is:
0: main(A,[]):-cardiac_cycle(A,[])
1: cardiac_cycle(A,B):-q(A,C),rs(C,D),st(D,E),t(E,F),
                       tp(F,G),p(G,H),pr(H,B)
2: q(A,B):-l(A,C),q(C,B)
3: q(A,B):-B=A
4: l(A,B):-negative_intermediate_slope(A,B)
5: l(A,B):-negative_large_slope(A,B)
7: rs(A,B):-c(A,C),d(C,D),e(D,B)
8-12: c(A,B):-large_slope(A,C),c(C,B)
9: c(A,B):-n(A,B)
...
```

The new program will not cover the four first negative examples, but only covers the fifth and the sixth ones. By repeatedly using the debugger we can find an other erroneous clause. The answer *no* given by the user to the question

```
Is it ok [d([ni,nl,l,ns,ns,...],[l,ns,ns,i,h,...])] (y/n) n
```

means that the third modification was recognized: we used $\texttt{d}--\texttt{>l}, \texttt{d}$ instead of $\texttt{d}--\texttt{>negative_large_slope}$.

```
 Checking positive examples:

 Checking negative examples:
56: main([ni,nl,l,l,nl,...],[]) covered.
57: main([nl,ni,l,l,nl,...],[]) covered.

 The above theory:
    cover   7 positive samples from 7 (100.00%) and
    fail on 4 negative samples from 6 (66.67%).

 The fact main([ni,nl,l,l,nl,...],[])  is covered by the theory.
 Starting the false proc. algorithm to determine the basis of the unfolding.

 Is it ok [l([ni,nl,l,l,nl,...],[nl,l,l,nl,ni,...])] (y/n) y
 Is it ok [l([nl,l,l,nl,ni,...],[l,l,nl,ni,nl,...])] (y/n) y
...
 Is it ok [d([ni,nl,l,ns,ns,...],[l,ns,ns,i,h,...])] (y/n) n
```

The next false clause was recognised. The clause found is spezialized by the unfolding. The resolvents of the clause are:

```
d(A,B):- negative_large_slope(A,C),d(C,B)
d(A,B):- negative_intermediate_slope(A,C),d(C,B)
d(A,B):- negative_small_slope(A,C),d(C,B)
```

In the derivation of positive examples the clauses

```
d(A,B):-negative_intermediate_slope(A,C),d(C,B)
d(A,B):-negative_small_slope(A,C),d(C,B)
```

do not occur, therefore they are removed from the set of clauses. The result of this unfolding is presented below.

```
 10: d([ni,nl,l,ns,ns,...],[l,ns,ns,i,h,...]):-
       l([ni,nl,l,ns,ns,...],[nl,l,ns,ns,i,...]),
       d([nl,l,ns,ns,i,...], [l,ns,ns,i,h,...])

- trying resolvent(s): [10-1] actual minimum is: 0.
- trying resolvent(s): [10-2] actual minimum is: 4.141709450076293.

 The result of the unfolding is:
0: main(A,[]):-cardiac_cycle(A,[])
1: cardiac_cycle(A,B):-q(A,C),rs(C,D),st(D,E),t(E,F),
                       tp(F,G),p(G,H),pr(H,B)
2: q(A,B):-l(A,C),q(C,B)
3: q(A,B):-B=A
4: l(A,B):-negative_intermediate_slope(A,B)
5: l(A,B):-negative_large_slope(A,B)
7: rs(A,B):-c(A,C),d(C,D),e(D,B)
8-12: c(A,B):-large_slope(A,C),c(C,B)
9: c(A,B):-n(A,B)
10-5: d(A,B):-negative_large_slope(A,C),d(C,B)
11: d(A,B):-l(A,B)
...

 Checking positive examples:

 Checking negative examples:

 The above theory:
    cover   7 positive samples from 7 (100.00%) and
    fail on 6 negative samples from 6 (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 6.3.

5 Conclusion, Further Work

In this paper an integrated system has been presented which can classify and learn ECG waveforms. Combining the IMPUT method with the PECG system the integrated system was able to suggest a correct clause for the user which replaced the buggy clause recognized during the debugging process.

In general the graphic viewer helps to find bugs, and in our case shows the ECG subpattern is currently being analyzed. This module was implemented in C, the C routines are UNIX portable and work together with the Prolog interpreter.

The main advantage of our system comparing with Skordalakis' approach is that our integrated tool can positively assists the developer (user) working with the ECG classifier program because the program itself is "self-correcting".

Applying the IMPUT learning method to the ECG domain we found that correct DCG programs can be inferred for the basic structures of complex ECG waveforms. In these approaches the user only needs to prepare very general initial DCG programs. The extended PECG system is able to specialise these programs into the correct ones in a very effective way. However, our general experiences has shown for us that this method is not really suitable for the learning rules which describe the computation of the values of the arguments of these structures. For this task we are going to try to apply other ILP techniques. In general it is concluded that the effective employment of the ILP method for real like applications may require the combination of different ILP techinques.

The current version of the PECG system is about 3500 lines in SICStus Prolog, and 500 lines written in C for the implementation of the graphic viewer. The IDT module is about 1400 Prolog lines, containing 150 predicates while the classifier program is more than 2000 lines in Prolog. The IMPUT system itself contains more than 800 lines without the IDT module. [5]

We are going to extend our system in the following way:

At present the PECG system does not contain any preprocessing module for the linguistic representation of ECG waveforms. Therefore we could test the performance of our system with only preprocessed data presented in the literature. To make our system a complete ECG recognizer we plan to implement these preprocessing modules. We also plan to ensure an interactive connection between the graphic viewer and the user. It means that the user may control the analyzing process from the graphic window by marking the subparts of the waveforms to be analyzed.

At present, the PECG system is able to analyse a large set of ECG waveforms. We plan to apply ILP methods to transformations of the classifier program, such that the transformed programs can recognize further ECG waveforms appearing in actual clinical situations. We are working on this task in conjunction with ECG medical experts from the Medical University in Szeged.

References

1. Alexin, Z., Gyimóthy, T., Boström, H.: Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner in Proceedings of the 12th European Conference on Artificial Intelligence ECAI-96 ed. Wolfgang Wahlster, Budapest, Hungary (1996) 403–407 John Wiley & Son's Ltd. 1996.
2. 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
3. De Raedt, L., Bruynooghe, M.: A theory of clausal discovery Proc. of the 13th International Joint Conference on Artificial Intelligence Morgan Kaufmann (1993)

[5] The implementation language currently is SICStus Prolog Version 3.0 #3 running on a SUN SPARCStation.

4. Deransart, P., Małuszyński, J.: Relating Logic Programs and Attribute Grammars Journal of Logic Programming 2, (1985) 119–156
5. Džeroski, S., Lavrač, N.: Inductive Learning in Deductive Databases, IEEE Transactions on Knowledge and Data Engineering 5 (6): (1994) 939–949
6. Fritzson, P., Gyimóthy, T., Kamkar, M., Shahmeri, N.: Generalized Algorithmic Debugging and Testing in Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario 1991. ACM SIGPLAN Notices 26, 6, (1991) 317–326
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. 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
9. Lavrač, N., Džeroski, S.: Inductive Logic Programming: Techniques and Applications Ellis Horwood, (1994)
10. Muggleton, S.: Inverse entailment and Progol, New Generation Computing Vol 13. (special issue on Inductive Logic Programming) Ohmsha, (1995) 245–286
11. Muggleton, S., De Raedt, L.: Inductive Logic Programming: Theory and methods, Journal of Logic Programming 19 (20) (1994) 629–679
12. Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM 31:6 June (1988) 676–686
13. 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
14. 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: 231-278, (1980)
15. Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)
16. Skordalakis, E.: ECG Analysis in Syntactic and Structural Pattern Recognition Theory and Applications ed. Bunke, H. and Sanfeliu, A. World Scientific (1990) 499–533
17. Skordalakis, E., Papakonstantinou G.: Towards an Attribute Grammar for the Description of ECG Waveforms 7th International Conference on Pattern Recognition (1984).
18. 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
19. Udupa J.K., Murthy I.S.N.: Syntactic Approach to ECG Rhythm analysis IEEE Transactions on Biomedical Engineering, vol. BME-27, No.7 pp 370-375, July 1980

6 Appendix

6.1 The syntactic part of the original description for a cardiac cycle of an ECG

```
main(Inp,[])    :-     cardiac_cycle(Inp,[]).
```

```
cardiac_cycle   -->    q, rs, st, t, tp, p, pr .

q               -->    l, q.
q               -->    {true}.

l               -->    negative_intermediate_slope.
l               -->    negative_large_slope.

rs              -->    c, d, e.

c               -->    large_slope, c.
c               -->    large_slope.

d               -->    negative_large_slope, d.
d               -->    negative_large_slope.

e               -->    large_slope, e.
e               -->    intermediate_slope, e.
e               -->    {true}.

st              -->    i,st.
st              -->    {true}.

i               -->    horizontal_slope.
i               -->    small_slope.
i               -->    negative_small_slope.

t               -->    f,g,h.

f               -->    k,f.
f               -->    {true}.

k               -->    intermediate_slope.
k               -->    small_slope.
k               -->    horizontal_slope.

g               -->    i,g.
g               -->    {true}.

h               -->    m ,g.
h               -->    {true}.

m               -->    negative_intermediate_slope.
m               -->    negative_small_slope.

tp              -->    i,tp.
tp              -->    {true}.

p               -->    t.
```

```
pr              -->    i, pr.
pr              -->    {true}.

negative_large_slope        --> [nl].
negative_intermediate_slope --> [ni].
negative_small_slope        --> [ns].
large_slope                 --> [l].
intermediate_slope          --> [i].
small_slope                 --> [s].
horizontal_slope            --> [h].
```

6.2 The input description for our system

```
main(Inp,[])    :-    cardiac_cycle(Inp,[]).

cardiac_cycle   -->    q, rs, st, t, tp, p, pr .

q               -->    l, q.
q               -->    {true}.

l               -->    negative_intermediate_slope.
l               -->    negative_large_slope.
l               -->    negative_small_slope.

rs              -->    c, d, e.

c               -->    n, c.
c               -->    n.

n               -->    large_slope.
n               -->    intermediate_slope.

d               -->    l, d.
d               -->    l.

e               -->    n, e.
e               -->    {true}.

st              -->    i,st.
st              -->    {true}.

i               -->    horizontal_slope.
i               -->    small_slope.
i               -->    negative_smal

t               -->    f,g,h.

f               -->    k,f.
f               -->    {true}.
```

```
k               -->    intermediate_slope.
k               -->    small_slope.
k               -->    horizontal_slope.

g               -->    i,g.
g               -->    {true}.

h               -->    m,h.
h               -->    {true}.

m               -->    negative_intermediate_slope.
m               -->    negative_small_slope.

tp              -->    i,tp.
tp              -->    {true}.

p               -->    t.

pr              -->    i, pr.
pr              -->    {true}.

background negative_large_slope        --> [nl].
background negative_intermediate_slope --> [ni].
background negative_small_slope        --> [ns].
background large_slope                 --> [l].
background intermediate_slope          --> [i].
background small_slope                 --> [s].
background horizontal_slope            --> [h].

positive main([ni,ni,l,l,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,
              ns,ns,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
positive main([nl,nl,l,nl,i,s,s,s,s,s,s,i,s,s,s,s,ns,ns,s,s,s,s,s,
              s,s,s,i,s,s,s,s,ns,ns],[]).
positive main([ni,nl,l,l,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,
              ns,ns,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
positive main([nl,ni,l,l,nl,nl,i,l,h,s,ns,h,s,ns,s,h,h,s,ns,ns,ni,
              h,s,ns,h,s,ns,h,s,s,h,h,s,ns,ns,ni,h,s,ns,h],[]).
positive main([ni,nl,ni,nl,l,l,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,
              ns,ns,ns,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
positive main([nl,nl,l,l,l,nl,i,l,i,ns,h,s,ns,h,s,h,i,ns,h,s,ns,ns,
              ns,h,s,ns,h,s,ns,h,s, h,i,ns,h,s,ns,ns,ns,h,s,ns] ,[]).
positive main([ni,nl,l,nl,nl,nl,l,i,l,h,s,ns,h,s,ns,h,s,h,s,ns,ni,ns,
              h,s,ns,h,s,ns,h,s,h,s,h,s,ns,ni,ns,h,s,ns,h],[]).

negative main([ns,ni,ni,l,l,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,
              ns,ns,ns,ns,ns,ns, ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
negative main([ns,ns,nl,nl,l,nl,i,s,s,s,s,s,s,i,s,s,s,s,ns,ns,s,s,s,s,
              s,s,s,s,i,s,s,s,s,ns,ns],[]).
negative main([ni,nl,l,l,nl,ni,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,
```

```
              ns,ns,ns,ns,ns,ns,ns, ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
negative main([nl,ni,l,l,nl,ni,ni,nl,i,l,h,s,ns,h,s,ns,s,h,h,s,ns,ns,
              ni,h,s,ns,h,s,ns,h,s,s,h,h,s,ns,ns,ni,h,s,ns,h],[]).
negative main([ni,nl,ni,nl,l,i,l,nl,l,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,
              ns,ns,ns,ns,ns,ns,ns,ns,ns,i,h,ns,ns,ns,ni,ns,s,s,s],[]).
negative main([nl,nl,l,l,i,i,l,nl,i,l,i,ns,h,s,ns,h,s,h,i,ns,h,s,ns,ns,
              ns,h,s,ns,h,s,ns,h,s, h,i,ns,h,s,ns,ns,ns,h,s,ns] ,[]).
```

6.3 The output of the extended PECG system

```
0: main(A,[]):-cardiac_cycle(A,[])
1: cardiac_cycle(A,B):- q(A,C),rs(C,D),st(D,E),t(E,F),
                        tp(F,G),p(G,H),pr(H,B)
2: q(A,B):- l(A,C),q(C,B)
3: q(A,B):- B=A
4: l(A,B):-negative_intermediate_slope(A,B)
5: l(A,B):-negative_large_slope(A,B)
7: rs(A,B):-c(A,C),d(C,D),e(D,B)
8-12: c(A,B):-large_slope(A,C),c(C,B)
9: c(A,B):-n(A,B)
10-5: d(A,B):-negative_large_slope(A,C),d(C,B)
11: d(A,B):-l(A,B)
12: n(A,B):-large_slope(A,B)
13: n(A,B):-intermediate_slope(A,B)
14: e(A,B):-n(A,C),e(C,B)
15: e(A,B):-B=A
16: st(A,B):-i(A,C),st(C,B)
17: st(A,B):-B=A
18: t(A,B):-f(A,C),g(C,D),h(D,B)
19: f(A,B):-k(A,C),f(C,B)
20: f(A,B):-B=A
21: g(A,B):-i(A,C),g(C,B)
22: g(A,B):-B=A
23: h(A,B):-m(A,C),h(C,B)
24: h(A,B):-B=A
25: m(A,B):-negative_intermediate_slope(A,B)
26: m(A,B):-negative_small_slope(A,B)
27: k(A,B):-intermediate_slope(A,B)
28: k(A,B):-small_slope(A,B)
29: k(A,B):-horizontal_slope(A,B)
30: i(A,B):-horizontal_slope(A,B)
31: i(A,B):-small_slope(A,B)
32: i(A,B):-negative_small_slope(A,B)
33: tp(A,B):-i(A,C),tp(C,B)
34: tp(A,B):-B=A
35: p(A,B):-t(A,B)
36: pr(A,B):-i(A,C),pr(C,B)
37: pr(A,B):-B=A
```

This article was processed using the LaTeX macro package with LLNCS style