Classifying ECG Waveforms in Prolog

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

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

Abstract. The basic idea of the presented system (PECG) is to integrate an ECG classifier program (implemented in Prolog) with the IDT algorithmic debugger module and a graphic viewer. If the ECG classifier cannot analyze the input then the user is helped by the built-in IDT (Interactive Diagnosis and Testing) debugger to find the place of the error. The IDT improves the Shapiro's original debugging method and can be used both in the debugging and in the testing of Prolog programs. In the PECG system the algorithmic debugger IDT has been extended with a graphic viewer. This graphic module shows the ECG waveform is being analyzed, in this way it gives an effective assistance to the user to answer the questions invoked by the debugger.

1 Introduction

In the civilized world, more and more people suffer from heart disease. In the diagnosis of heart diseases the ECG is one of the simplest method. ECG is the graph of the potential difference induced by the heart muscle activity. These waveforms give a good characterization of certain illnesses. Finding the abnormal form among the large number of regular waveforms demands a lot of time and attention from a cardiologist. There are many ECG processing systems which use computers to recognize abnormal waveforms [2], [8], [15].

Among them there are systems which based on the attribute grammars. These systems are successful because they can take into account both the syntactical characteristics and the numerical parameters as well. Recognizing ECG waveforms with attribute grammars is still recently investigated. Since in the case of one patient the number of the abnormal waveforms among the investigated waveforms could be very few it is effective to make such a system which classifies the recognized ECG waveforms. Therefore the doctor can easily select the abnormal waveforms. The classifying process can be done according to some synthesized attributes and empirically established diagnostic criteria.

In this paper we present a system (called PECG) which is an ECG waveform classifier. The system contains three main modules the *ECG processing*, the *IDT* and the *graphic viewer* modules. Its theoretical basis is an attribute grammar approach of ECG waveform analysis published by Skordalakis et al [14].

This system can classify the recognized ECG waveforms according to some synthesized attributes. It contains a preprocessing part which produces a linguistic representation from an ECG signal. This linguistic representation is the input of the PECG system.

The basic idea of our system is to integrate the ECG classifier program (implemented in Prolog) with the IDT algorithmic debugger module and a graphic viewer. This integrated tool can recognize if any change 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 [13] and can be used both in the debugging and in the testing of Prolog programs [6], [7], [11].

Shapiro's algorithmic debugger method can isolate an erroneous procedure (clause) given a program and an input on which it behaves incorrectly. Shapiro's model has been applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. A major drawback of this debugging method is the great number of queries to the user for the correctness of intermediate result of procedure calls. The main concept of our improvement in the IDT is the following. If the program has already been tested by the Prolog version of the Category Partition Method (CPM) [10] then the test results for the procedures can be used in the debugging phase to answer the questions.

In the PECG system the algorithmic debugger IDT has been extended with a graphic viewer. This graphic module shows the ECG waveform is being analyzed, in this way it gives an effective assistance to the user to answer the questions invoked by the debugger.

The results achieved in the system PECG and presented in this paper can be summarized as follows:

- a 'large' (more than 2000 lines) Prolog program has been implemented for the classification of ECG waveforms. The Prolog program can recognize a large class of ECG waveforms.
- The classifier program has been integrated with the IDT algorithmic debugger module developed by the authors.
- In addition a graphic viewer module is implemented, which helps the users during the debugging and makes the processing of ECG waveforms more transparent.

The main advantage of the integrated system PECG is that if the classifier program could not recognize an input correctly, then the clause which is responsible for the incorrect behavior can be identified using the IDT and the graphic viewer parts of the system. The PECG system is implemented in SICStus Prolog. [4]

In the Section 2 the structure of the whole PECG system is presented. Section 3 contains the description of the recognition of ECG waveforms. In the Section 4 we write about the logical approach for representing of ECG. Section 5 deals with IDT module. Section 6 describes the analysis of ECG waveform with the system. The last Section contains conclusion and further work.

2 The structure of the PECG system

The structure of the PECG 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 based on the Shapiro's Interactive Diagnosis Algorithm [13] and the Category Partition Method [10].
- *the graphic viewer module:* this part makes visible the current state of the debugging.

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

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

The task of the above parts is the following:

- *primitive pattern selection:* 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).
- *primitive pattern extraction:* the main task of this subpart is the extraction of the peaks. This part is based on the recognition of the noisy peaks. The real peaks are the difference between the set of all peaks and the noisy ones.
- *the linguistic representation:* the result of the last two steps can be represented as a string of symbols from an alphabet.
- *the ECG classifier:* this module recognizes the ECG waveforms from their linguistic representations.

The IDT part can be divided into three submodules:

[4] The implementation language currently is SICStus Prolog Version 2.1 #9 running on a SUN SPARCStation.

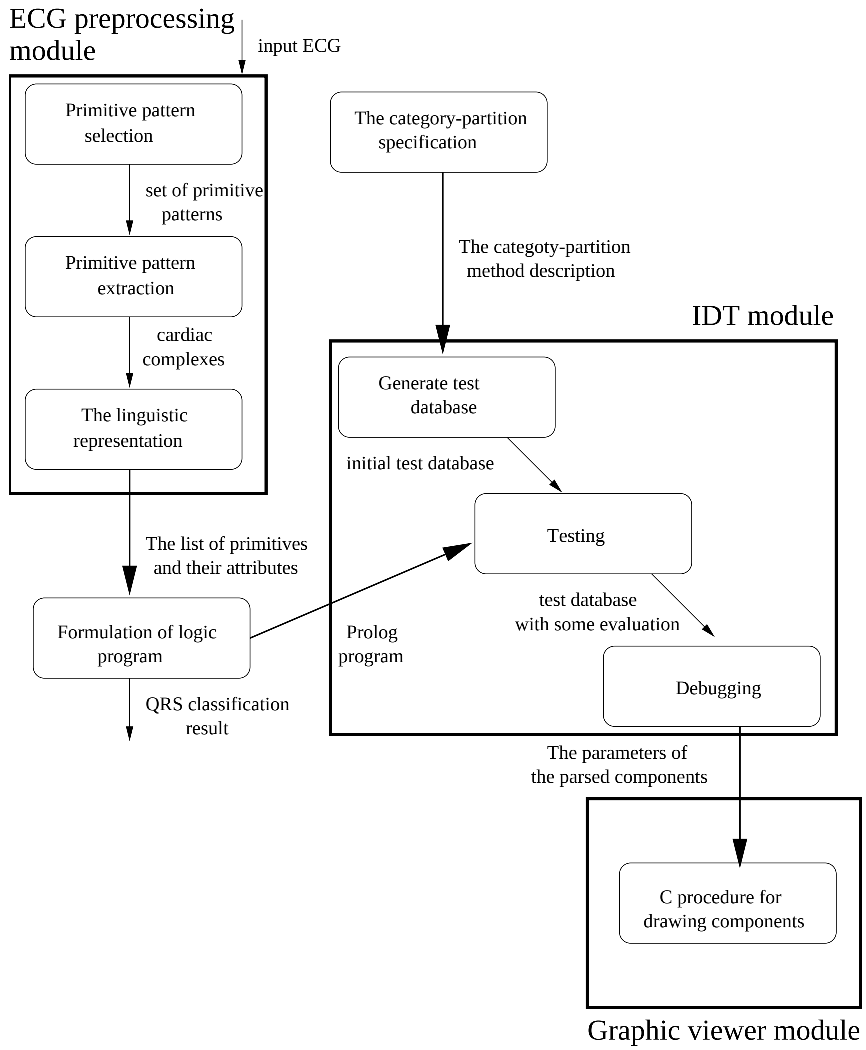

Fig. 1. The structure of the PECG system

- *generate test database:* from a CPM description the program generates the initial test database.
- *testing:* a predicate can be chosen and the user can give a representing element of its testframe generated for this predicate and has an undefined evaluation. Then he or she can decide whether for the given input the output correct or not.
- *debugging with IDT method:* in this part the main idea is that if the program has already been tested then the test results can be directly applied without

asking the user "difficult" questions.

The graphic viewer module helps the users to decide whether the program with a given input works correctly or not. It illustrates the steps of debugging phase. At the beginning of the debugging the whole ECG waveform will be displayed and in each step of the debugging the current component of the input will be presented. This module contains C routines.

3 Recognition of ECG Waveforms

In this section we give a short overview on the syntactic recognition of the ECG waveforms. The methodologies and notations are borrowed from the Skordalakis' paper [14]. The electrocardiograms are largely employed as a diagnostic tool in clinical practice. With the help of such diagnostic information certain heart abnormalities can be recognized.

Several methods have been used in the recognition and analyzing ECG signals but in every case there are two phases of these processes:

(1) in the "pattern recognition and measurement phase" certain patterns on the ECG are recognized and several parameters (amplitude and duration) are measured to characterize the waves
(2) in the "interpretation phase" with the help of the patterns recognized in the first phase and the 'attributes' which belong to every primitive pattern we can make a diagnosis. In this phase we can use some empirical diagnostic criterias established by analyzing of millions of ECG's.

Nowadays many systems are worked out with the goal to automating this process [14], [8]. In Figure 2. the three complexes (P, QRS, T) are presented which can be observed as an ECG waveform. Every cardiac complex contains one or more cardiac waves, but the most important phase in the heart action is the QRS complex. The basic parts of the ECG's are the waves. A segment can build up from waves. An interval contains one or more segments. The set of intervals gives us the complex and a cardiac cycle can be decomposed on intervals. For example we can compose the Q, R, and S waves into an QRS complex. The other patterns in a cardiac cycle are the interwave segments (PR, ST, TU, UP, TP) and the intervals (PR, QT, RR, PP) as can be seen in Figure 2. The parameters amplitude and duration can be used in the complexes while segments and intervals contain duration only.

In the syntactic recognition of ECG there are four subproblems:

(1) primitive pattern selection
(2) primitive pattern extraction
(3) primitive pattern encoding
(4) formulation of a pattern grammar

For the first three subproblems details can be found in [14]. In this paper we will concentrate on the analysis problem of 'pattern grammar'. We use three

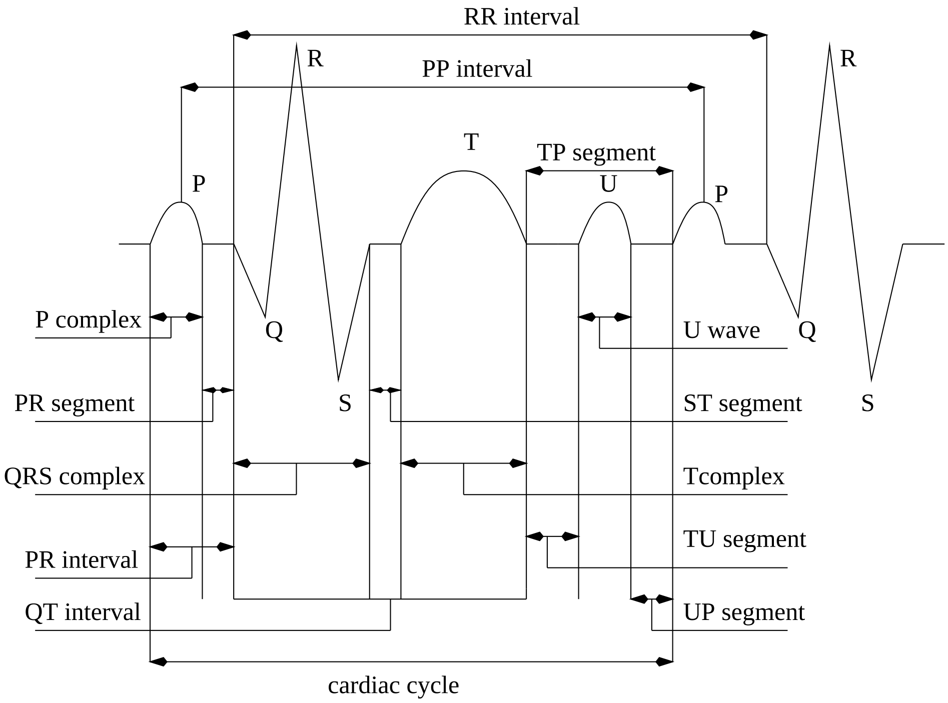

Fig. 2. The components of an ECG waveform

primitive patterns "peak", "straight line segment", and "parabolic segment". In the electrocardiographic pattern we can separate two main shapes the complex and the segment. A complex builds up from series of peaks. The segment can be divided into two other classes which are the straight line and the parabola.

A peak pattern can be seen in Figure 3. This is the most important part of the signal and it can be characterized by the "left peak boundary", the "peak extremum" which is the highest point in the peak if peak is positive and the lowest if it is negative and the "right peak boundary". We call the points between the left boundary and the peak extremum "left arm" of the peak and the "right arm" of the peak are the points between the peak extremum and the right peak boundary. A peak can be described with the help of twelve attributes which can be found in the Appendix 1. The peaks are labeled with $P_1, P_2, \ldots$. The straight line segment k has six and the parabolic segment K has seven attributes which can be also seen in the Appendix 1.

4 Logical Approach for Representation of ECG

In the paper [14] an attribute grammar was used for the recognition of ECG waveforms. The ECG waveform was described as a symbol from the alphabet

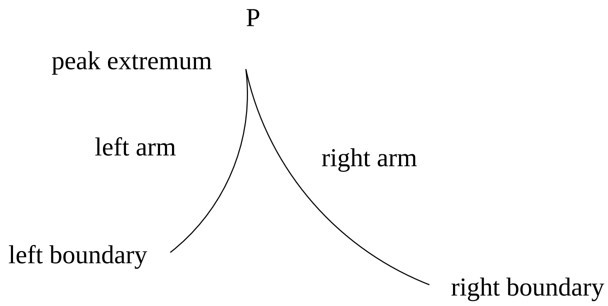

Fig. 3. A peak pattern

$\Sigma = \{K^+, K^-, E, \Pi\}$ where K^+ and K^- represent the positive and negative peaks, E the straight line segment and Π the parabolic segment.

Our approach is based on the method presented in [14] but we use logic program representation for the recognition of the categories. The attribute grammar presented in [14] was used as a starting-point and it was rewritten into Prolog. The difference between an attribute grammar representation and a Prolog program made the rewriting process non-trivial. We used the definite clause grammar (DCG) [12] representation because this notation gives a good tool for the rewriting attribute grammars [3]. The rewriting process is the following:

(1) the nonterminal symbols of the grammar correspond to predicates in the Prolog program
(2) both the inherited and the synthesized attributes turn into the arguments of the Prolog predicates. The twenty most frequently used arguments and their meaning can be seen in Appendix 3.
(3) The semantic routines and some necessary auxiliary routines are converted into extra conditions in the form of Prolog procedure calls. They may be included in the right hand side of a grammar rule.

QRS detection and recognition. There are 13 alternatives of Prolog clauses to recognize a series of consecutive peaks as QRS complex. A QRS complex can be built up from one to seven peaks and these clauses match the QRS pattern specified by their morphology. The question: "whether a sequence of peaks belongs to the QRS class or not", can be decided with the help of threshold values. The following conditions must be satisfied:

(1) the angle between the right arm of peak i and left arm of peak $i+1$ is less than ε_2
(2) the angle of each peak is less than ε_5
(3) the energy of at least one peak is greater than ε_1.

P, T detection and recognition. P or T complexes can be found only between two QRS complexes. It means that complexes are not detected before the first

QRS complex and after the last QRS complex. P or T complexes consist of one or two consecutive peaks and they are discriminated from other peaks with the help of semantic routines using the energies of the peaks. The peaks are recognized as P or T complex when the height and the duration of each arm in every peak are greater than ε_3 and ε_4, respectively.

QRS classification We need to establish the distance between a QRS complex and a class of QRS complexes or the distance between two QRS complexes. In the first case the distance for the QRS complex and each QRS complex in the classes is computed. In the second case the *city block* metric helps us to establish the distance between two QRS complexes. The normalized duration and height of each wave component of the QRS complex are used to compute this distance.

The syntactic part of the original attribute grammar is given in the Appendix 2. There were some problems which we had to solve in the rewriting process. The implementation of the semantic functions was complicated because it was written for an imperative and not for a logic programming language. We had to handle a large number of parameters.

5 IDT System

In this section the IDT algorithmic debugger system is presented. The IDT system can be used both in the debugging and in the testing of Prolog programs. The basic idea of IDT is that the test database prepared by the Prolog version of the Category Partition Method (CPM) can be used to reduce the amount of user interactions during the debugging. In the IDT system the diagnosis process effects the testing hence IDT can be considered as an integrated debugging and testing method. In the rest of this section first an overview of the CPM introduced in [10] is given than we briefly describe the Shapiro's debugging method presented in [13] and explain how the basic *single-stepping* debugging algorithm can be modified with the help of the CPM method [6], [7], [11].

5.1 The CPM Testing Method

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

The categories can be divided into disjoint classes — called *choices* — presuming that all the elements within a choice have an identical observable behavior with respect to correctness. When the categories and choices for a program have been defined, then all the possible *test frames* can be generated. Intuitively, a test frame expresses a certain syntactic property of input elements with the same semantic behavior, expressed as a combination of input properties. Hence, a test frame covers exactly one choice from each category. In general, there may be many superfluous test frames among the generated ones, that is, property combinations that are not sensible. These frames can be eliminated by associating *selector expressions* with the choices. A choice can be included in a test frame if the associated selector expression is *true* with respect to the combination of choices for the frame.

The selector expressions make use of *property names*. A property is also associated with a choice and can be considered as a Boolean variable. The value of this variable is *true* if the given frame contains that choice. An extended version of the CPM method is presented in [4].

5.2 Shapiro's Method for Algorithmic Debugging

Shapiro's algorithm can isolate an erroneous procedure (clause), given a program and an input on which it behaves incorrectly. The algorithm traverses the proof tree of a program and asks the user about the expected behavior of each resolved goal. The user has to verify the correctness of each goal, in which case a bug inside a certain procedure can be identified.

The notations that are needed for understanding this part are the following. P will denote the *logic program* and M its *interpretation* (that is, intended meaning). T will denote a *proof tree* in the conventional sense where each node stands for a resolved atom, and each subtree with $p(\ldots)$ as the root and $p_1(\ldots),\ldots,p_n(\ldots)$ as its sons corresponds to the clause $p(\ldots) :- p_1(\ldots),\ldots,p_n(\ldots)$ in the program P.

Let p be a predicate that terminates on input x and returns y as output such that $< p,x,y > \notin M$. This implies that the program P has at least one incorrect clause. For finding it, the proof tree T rooted by $< p,x,y >$ is traversed in a postorder manner. During the traversal over T, at each node $< q,u,v >$ a question is issued whether or not $< q,u,v > \in M$. Let us suppose that the first negative answer is received at the node $< q,u,v >$. Let the direct descendants of $< q,u,v >$ in T be $< q_1,u_1,v_1 >,\ldots,< q_m,u_m,v_m >$. Since a postorder traversal strategy was applied, it holds that $< q_i,u_i,v_i > \in M$ for all i $(1 \leq i \leq m)$. From this follows that the clause $q(\ldots) :- q_1(\ldots),\ldots,q_m(\ldots)$ covers the triple $< q,u,v >$ which is not in M. The algorithm stops at the node $< q,u,v >$ and returns the clause instance $< q,u,v > :- < q_1,u_1,v_1 >,\ldots,< q_m,u_m,v_m >$. By this method an erroneous clause can always be identified assuming that the answers to the membership questions are correct.

5.3 IDT — Integration of Shapiro's Method with CPM

The idea of the integration is that the algorithm does not ask the user about the correctness of each resolved goal if it finds the evaluation of this goal in the database for test frames. To implement the CPM and the Shapiro's Single Stepping algorithm first an initial test description is prepared from which a test database is generated by the Prolog program. The input of the program is a test specification which is given in a Prolog form. The descriptions are Prolog facts in the following form:

choice_of($category\_name_1$,$choice\_name_1$).
choice_of($category\_name_1$,$choice\_name_2$).

choice_of($category\_name\_n$,$choice\_name_{n_1}$).
choice_of($category\_name\_n$,$choice\_name_{n_2}$).

In the IDT system we can choose a function from the following menu:

```
  1. Load CPM description.
  2. Listing CPM description.
  3. Generate test database.
  4. Listing test database.
  5. Clear test database.
  6. Testing.
  7. Debugging with Shapiro's method.
  8. Debugging with IDT method.
  9. Load test database.
 10. Save test database.
  0. Quit.

 Enter selection:
```

Only the functions 3, 6, 8 are explained. The meaning of the other functions is obvious.

3. Generate test database

From the loaded CPM description the program generates the initial test database. The elements of test database are Prolog facts in the following form:

```
test (P, A, F, IO, E).
```

Where P is the predicate name, A is the predicate arity, F is the test frame (represented by a list of corresponding choices), IO is the list of input and output values of representing elements, E is the evaluation of the test frame. Every fact begins with functor *test*. It followed by the name of the predicate which will be tested and its arity. The third parameter is the description of the test frame. The next parameter contains the list of pairs of input/output parameters on which this clause will be tested (initially it is an empty list). The last argument of the item will be the evaluation of the test (initially undefined). The test database is manipulated by the Prolog system predicates *retract* and *assert*.

In the IDT system searching functions have been introduced to decide which test frame contains the given input. For every choice name a searching function can be defined. If the searching function is not defined the selection of the proper testframe is performed via menu. The searching functions are Prolog predicates. For example the searching function for choice name zero can be defined as follows:

```
zero_waves_qrs(qrs,11,_,0,_,_,_,_,_,_,_,_,_).
```

6. Testing.

At the beginning of testing the database does not contain any test case. The user can choose from a menu a predicate name which he or she wants to test and after it on the screen can be seen the list of testframes which have undefined evaluation. The user can choose one of them and can give a representing element of this frame. It means that the user types in the values of the input variables. Then it can be decided whether the evaluation that takes the input and returns the output is correct or not. If the given example does not belong to the selected frame the user can see a warning message on the screen and it is possible to repeat the process from the beginning. The user need not to fill in the full database because there are some items which are not relevant for the testing. The test database contains every proper combination of choices.

8 Debugging with IDT method.

The use of this part is explained in the next section.

6 Analyzing ECG Waveforms using IDT — An Example

In this section we explain how we can use the IDT system to debug the description of the ECG. The debugging process is supported by a graphical representation to trace the history of traversing the proof tree of the ECG program.

To demonstrate the work of our integrated debugger IDT, it is supposed that there is a bug in the logic representation of the QRS complex. For this reason one correct clause from the clauses of the QRS complex is removed. The IDT system is used to identify this bug namely to find that an QRS class is absent in the classification.

In the developing phase it was recognized that it is very difficult for the user to give answer for the questions asked by the system only from the string representation of a subpart of an ECG waveform because it contains a lot of parameters. Therefore a graphic viewer module has been implemented to make easier the task of the user. On the graphical implementation the user can follow which part of the ECG waveform is being analyzed.

It can be seen that it is very difficult to answer the questions during debugging because the predicates may contain sophisticated data structures as arguments. Many of these arguments are not relevant for the decision. To solve this problem first the following line was introduced in the test specification:

```
type (the_name_of_the_predicate(+, - ...)).
```

This line can be written for each predicate name which can be found in the program but it is not necessary. The meaning of this line is the following: with the help of the sign plus or minus the user can make a decision whether for the given predicate name its arguments will be turn out on the screen in the debugging phase or not. The default meaning can be defined with the line:

```
globaltype(all).
```

This means that for all predicates all arguments will be displayed. In the following a short example is presented. This example demonstrates the processing of the system. An ECG waveform input is given that is not recognized correctly by the system. Therefore ECG classifier is needed to be modified.

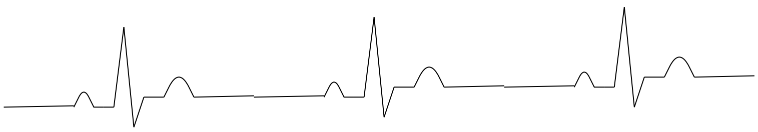

Fig. 4. An initial ECG waveform

Let us suppose that we have the following initial ECG waveform presented in Figure 4. 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 this example by means of the debugger it is realized that one possibility of the correction is a slight modification in the ECG syntax. A new derivation rule is needed to be added. For the better understanding the following notations are used:

- *n_peak* denotes a peak with negative peak extremum and eight parameters,
- *p_peak* denotes a peak with positive peak extremum and eight parameters,
- *line* denotes a straight line primitive segment with six parameters,
- *par* denotes the fourth kind of the primitive pattern the parabola with seven parameters.

First the user consults the system and if he or she gives the predicate *go* the main menu can be seen.

```
 1. Load CPM description.
 2. Listing CPM description.
 3. Generate test database.
 4. Listing test database.
 5. Clear test database.
 6. Testing.
 7. Debugging with Shapiro‘s method.
 8. Debugging with IDT method.
 9. Load test database.
10. Save test database.
 0. Quit.

Enter selection:
```

The user chooses the menu 1 then the name of the input Prolog file is typed in which contains a program for classifying the ECG waveforms.

```
Enter file name to load: ecg.pl

--> Load CPM description from file: ecg.pl.
```

The initial database can be generated with the help of the menu 3 from a CPM description. A simple CPM specification for the predicate `qrs` is listed below. It is followed by the initial test database for this predicate generated from the CPM description. Naturally, similar CPM test specifications can be prepared for other predicates as well e.g. `cardiac_cycles`, or `ecg_lead`. The description begins with the line `type(qrs(...))` which defines whether a parameter of qrs is printed out or not.

The category-partition specification for this predicate contains three categories `number_of_the_waves`, `number_of_the_classes`, `number_of_the_qrs`. Each category can be divided into two disjoint classes. For example the category `number_of_the_qrs` expresses the view that no QRS detected (choice `zero_qrs`, property `p_zero`), QRS detected (choice `non_zero_qrs`). The searching functions are simple Prolog facts. For the choice `non_zero_qrs` no searching function is given. In this case the user can choose this choice from a menu. The properties and the conditions e.g. `p_zero` reduce the testing space by excluding redundant combinations of input properties such as `[non_zero_classes, zero_qrs, non_zero_waves]`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       test description:                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

type(qrs(+,+,+,+,+,+,+,+,+,-,-)).

%       first category: number_of_the_waves
 choice_of(qrs,11,number_of_the_waves ,zero_waves).
 choice_of(qrs,11,number_of_the_waves ,non_zero_waves).

%       second category: number_of_the_classes
 choice_of(qrs,11,number_of_the_classes ,zero_classes).
 choice_of(qrs,11,number_of_the_classes  ,non_zero_classes).

%       third category: number_of_the_qrs
 choice_of(qrs,11, number_of_the_qrs ,zero_qrs).
 choice_of(qrs,11, number_of_the_qrs ,non_zero_qrs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       searching functions:                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
zero_waves(qrs,11,_,0,_,_,_,_,_,_,_,_,_).
non_zero_waves(qrs,11,_,Sw1,_,_,_,_,_,_,_,_,_) :- Sw1 \== 0.
zero_classes(qrs,11,_,_,_,0,_,_,_,_,_,_,_).
non_zero_classes(qrs,11,_,_,_,Sc0,_,_,_,_,_,_,_) :- Sc0 \== 0.
zero_qrs(qrs,11,_,_,_,_,_,[0],_,_,_,_,_).
undefined_search(non_zero_qrs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       properties:                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

p_zero(qrs,11,zero_qrs_qrs,X):-
member(zero_qrs_qrs,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       conditions:                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cond(qrs,11,zero_classes_qrs,X):-
 !,
 p_zero(qrs,11,_,X).

cond(qrs,11,zero_waves_qrs,X):-
 !,
 p_zero(qrs,11,_,X).
```

The generated test frames for all the admissible choice combinations from the three categories are:

```
Enter selection:  3

 The generated frames for qrs/11 are:

  [non_zero_classes,non_zero_qrs,non_zero_waves]
  [non_zero_classes,non_zero_qrs,zero_waves]
  [non_zero_classes,zero_qrs,non_zero_waves]
  [non_zero_classes,zero_qrs,zero_waves]
  [zero_classes,non_zero_qrs,non_zero_waves]
  [zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,zero_qrs,non_zero_waves]
  [zero_classes,zero_qrs,zero_waves]

 The screened frames for qrs/11 are:

  [non_zero_classes,non_zero_qrs,non_zero_waves]
```

```
  [non_zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,non_zero_qrs,non_zero_waves]
  [zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,zero_qrs,zero_waves]

Press <Enter> to continue ...
```

The menu 4 lists the current test database.

```
The list of current test database:

test(cardiac_cycles, 8, [non_zero1,non_zero2,non_zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,non_zero2,zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,non_zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,zero], [], undefined).
test(cardiac_cycles, 8, [zero1,zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,zero], [], undefined).
test(ecg_lead, 5, [zero1,zero2,zero], [], undefined).
test(qrs, 11, [non_zero_classes,non_zero_qrs,non_zero_waves], [], undefined).
test(qrs, 11, [non_zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,non_zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,zero_qrs,zero_waves], [], undefined).

Press <Enter> to continue ...
```

Choosing the menu 6 the user can execute the testing process.

```
 Enter selection:  6

    1. Frame: "cardiac_cycles,8".
    2. Frame: "ecg_lead,5".
    3. Frame: "qrs,11".
    0. Quit.

    Enter selection: 1
```

The debugger lists those predicate names that have CPM description. First the user should choose one from the predicate names. Then the program makes a list selecting the lines from the database with the chosen predicate name and undefined evaluation.

```
The frames for cardiac_cycles/9 are:
```

```
1. Frame: "[non_zero1,non_zero2,non_zero]".
2. Frame: "[non_zero1,non_zero2,zero]".
3. Frame: "[non_zero1,zero2,non_zero]".
4. Frame: "[non_zero1,zero2,zero]".
5. Frame: "[zero1,zero2,zero]".
0. Quit.

Enter selection: 1
```

Upon the user selects a frame then the next step is to give an example which falls into the selected frame.

```
Enter clause: cardiac_cycles(0,0,[0],0,0,[0],p_peak,par,line,p_peak,
n_peak,line,p_peak,par,p_peak,line,p_peak,n_peak,par,
p_peak,par,p_peak,par,p_peak,n_peak,line,p_peak,par],
[p_peak,par,line,p_peak,par,p_peak,line,p_peak,n_peak,
par,p_peak,par,p_peak,par,p_peak,n_peak,line,p_peak,par])
```

The most complex part of the system is the algorithmic debugging part. There are some built-in facilities that make the debugging process easier:

- *Debugging(break) points*: Here the user gives the list of predicate names where he wants that the debugger stops the execution of the program.
- *Debugging point [step, skip]*: The user has two possibilities either skip to the next breakpoint or switch to the trace mode. Trace mode means that the debugger executes the statement one-by-one and stops after each statement. This action will be performed when the debugger reaches one of the breakpoints.
- *Enter predicate*: That will be the goal and the debugger will process the proof tree of this goal.

```
--> Debugging.

Debugging points: segment, negative_peak, cardiac_cycles.

Debugging point [step, skip]: skip

Enter predicate: main(e).
```

The system executes the input Prolog program and displays the result. The user can make a decision whether it is correct or not. In this case the decision is that it is not correct. In the same time the graphic viewer shows the whole waveform (see Figure 5).

```
NUMBER OF CLASSES =  1
CLASS [1]: NUMBER OF QRS  = 3
QRS 1  BEGIN = 253   END = 280     MORPHOLOGY: r
```

```
QRS 2  BEGIN = 737   END = 764     MORPHOLOGY: r
QRS 3  BEGIN = 1224  END = 1250    MORPHOLOGY: r

 Is it correct main(e) (y/n)n
```

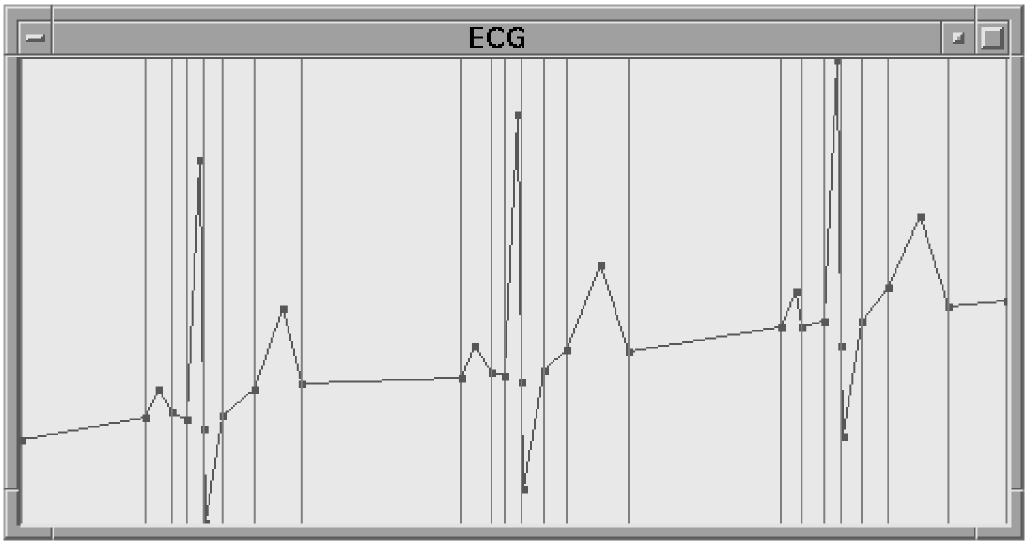

Fig. 5. The representation of the input ECG

After that the debugging phase is begun. 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 system displays the false clause. In the opposite case the debugging goes on with the next predicate. In this example the parameters are not displayed for the predicate `segment` because the test description contains the line `type(segment(-,-))`. The graphic viewer shows to the user the current subpattern displaying it in different line style see Figure 6, 7, 8.

```
Is it ok [segment( ? , ? )] (y/n)y
```

The debugger does not ask the user whether the predicate `cardiac_cycles` correct or not because the system can identify a testframe *[zero1,zero2,zero]* . This frame was tested during the testing process and the user gave as evaluation *correct*.

```
Is it ok [segment( ? , ? )] (y/n) y
```

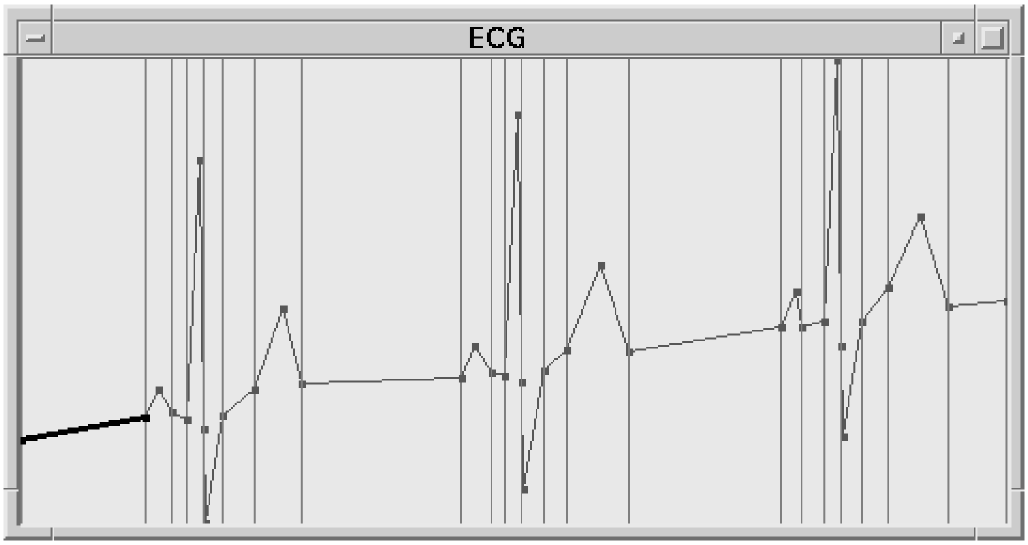

Fig. 6. The graphical displaying the actually asked primitive

There is no line `type(negative_peak())` for the predicate *negative_peak* therefore in this case the `globaltype(all)` is valid. It means that the whole list of the parameters is displayed.

```
Is it ok[negative_peak([n_peak,line,p_peak,par,p_peak,line,
p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,line,
p_peak,par],[line,p_peak,par,p_peak,line,p_peak,n_peak,par,
p_peak,par,p_peak,par,p_peak,n_peak,line,p_peak,pa])] (y/n)
```

If there are no searching functions for all the choices of the current predicate (there is a line in the test description *undefined_search(choice_name)*) then the user has to select the corresponding choice manually in the debugging process. The debugger lists the possible choices from which the user chooses one. The debugger executes an additional checking for the selection if it is a correct searching function for the selected choices. It can be seen in the Figure 9 that graphic viewer draws out the whole recognized QRS waveform in other graphic style.

```
Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )

    Category :  number_of_the_classes
    1 Choice: "non_zero_classes".
    2 Choice: "zero_classes".
    0. Quit

    Enter selection:   1
```

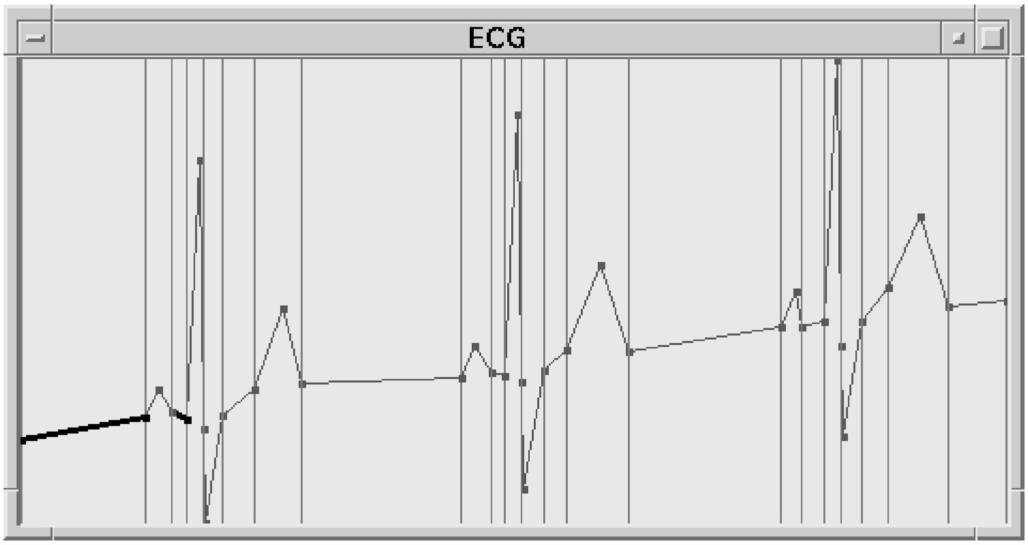

Fig. 7. The graphical displaying the actually asked primitive

```
 Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )

    Category :  number_of_the_qrs
    1 Choice: "non_zero_qrs".
    2 Choice: "zero_qrs".
    0. Quit

    Enter selection:  1

WARNING: Not existing searching function for non_zero_qrs_qrs

Press <Enter> to continue ...

Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )

    Category :  number_of_the_waves
    1 Choice: "non_zero_waves".
    2 Choice: "zero_waves".
    0. Quit

    Enter selection: 1

Is it ok [qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )] (y/n) n
```

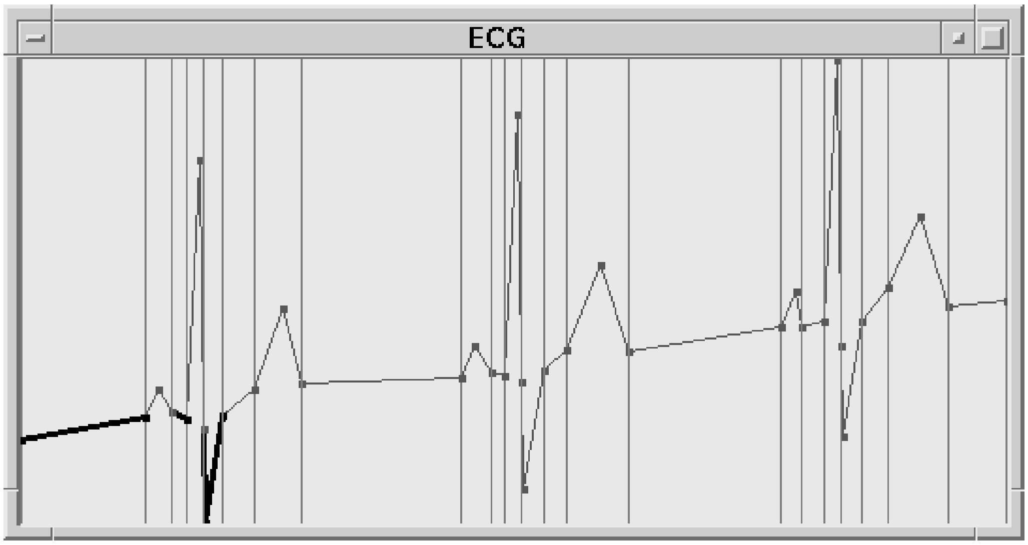

Fig. 8. The graphical displaying the actually asked primitive

The user answers the above question with *no*. It means that the system finds the false clause which will be displayed with the whole parameter list.

```
The false clause:
qrs(0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,p_peak,par,
    p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
    n_peak,line,p_peak,par],[n_peak,line,p_peak,par,p_peak,
    line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,
    line,p_peak,par]):-
write( qrs 12 ),nl,
insert_last(0,[],[0]),
insert_last(0,[],[0]),
insert_last(0,[],[0]),
retract(user:build_draw_qrs([])),
assert(user:build_draw_qrs([]))),
r(253,280,275,-220,-246,461,158466,11.1,[p_peak,n_peak,
  line,p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,
  par,p_peak,par,p_peak,n_peak,line,p_peak,par],[n_peak,
  line,p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,
  par,p_peak,par,p_peak,n_peak,line,p_peak,par]),
insert_last(158466,[],[158466]),
insert_last(11.1,[],[11.1]),
insert_last(253,[0],[0,253]),
```

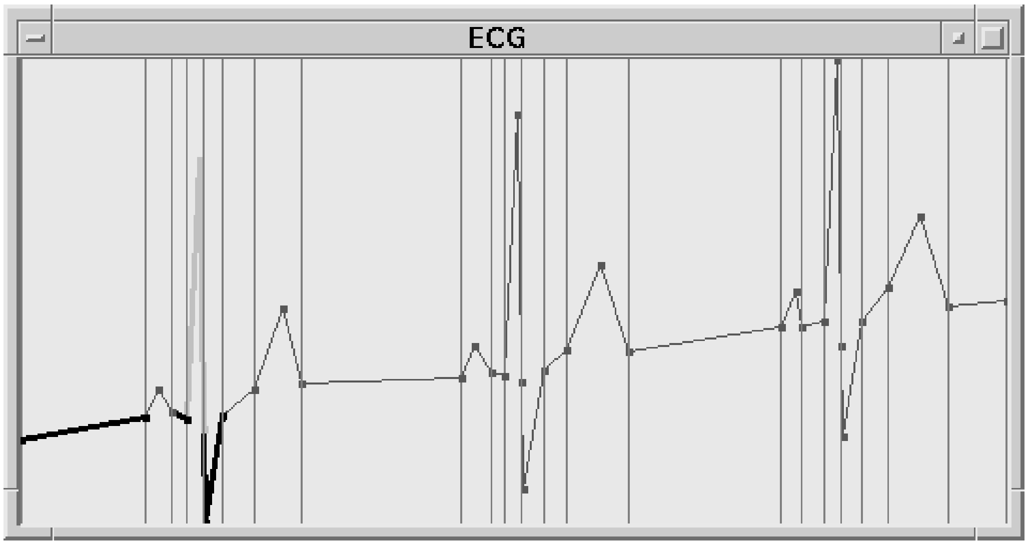

Fig. 9. The graphical representation of the recognized QRS part

```
insert_last(280,[0],[0,280]),
insert_last(461,[0],[0,461]),
12=12,
build_qrs(253,-220,275,461,280,-246),
sem_rou1(-220,[0],0,0,1,253,253,1,[1],280,280,1,1,
 [158466],[11.1],[],[0,253],[0,280],[0,461]).


Press <Enter> to continue ...
```

The debugging process finished. If the user chooses the menu 4 the system lists the test database. It contains pairs of input output parameters and the evaluation prepared during the debugging process.

```
 The list of current test database:

test(cardiac_cycles, 8, [non_zero1,non_zero2,non_zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,non_zero2,zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,non_zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,zero], [], undefined).
```

```
test(ecg_lead, 5, [zero1,zero2,zero], [], undefined).
test(qrs, 11, [non_zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,non_zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,zero_qrs,zero_waves], [], undefined).
test(cardiac_cycles, 8, [zero1,zero2,zero],
[([0,0,[0],0,0,[0],[p_peak,par,line,p_peak,n_peak,line,p_peak,
par,p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
n_peak,line,p_peak,par],[p_peak,par,line,p_peak,par,p_peak,
line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,
line,p_peak,par]])], correct).
test(qrs, 11, [non_zero_classes,non_zero_qrs,non_zero_waves],
[[0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,p_peak,par,
p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
n_peak,line,p_peak,par],[n_peak,line,p_peak,par,p_peak,line,
p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,line,
p_peak,par]],[0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,
p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,
par,p_peak,n_peak,line,p_peak,par],[n_peak,line,p_peak,par,
p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
n_peak,line,p_peak,par]])], incorrect).

Press <Enter> to continue ...
```

7 Conclusion, Further Work

In this paper the PECG system has been presented which can classify ECG waveforms. The classifier program is based on an attribute grammar description of ECG published by Skordalakis et al [14]. The basic idea of our system is to integrate the ECG classifier program (implemented in Prolog) with the IDT algorithmic debugger and the graphic modules. This integrated tool can recognize if any change in the classifier program is needed.

The IDT module improves Shapiro's original algorithmic debugging method by reducing the number of questions to the user. The validity of results of a procedure call is not asked from the user if it can be inferred from a CPM test configuration.

The graphic viewer also helps to find the bug. It shows the ECG subpattern is being analyzed currently. This module was implemented in C. The C routines are portable in UNIX and work together with the Prolog interpreter.

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 and the classifier program is more than 2000 lines in Prolog.

The main advantage of our system comparing with Skordalakis' approach is that our integrated tool could give an effective assistance to the developer (user) of the ECG classifier improving itself. We are going to extend the PECG system in the following directions:

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 only with preprocessed data presented in the literature. To make our system a complete ECG recognizer we plan to implement these preprocessing modules.

We are working on the improvement of the IDT module. A static slicing algorithm for Prolog programs has been developed and implemented in a joint work with the Helsinki University [5]. Using this slicing method further improvement can be achieved in reducing user interactions in the debugging process. The integration of the slicing algorithm into the IDT module is under development.

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.

The most significant improvement of the PECG system would be the integration of this system with the IMPUT interactive learning system developed by as a joint work of authors and a researcher from the Stockholm University [1]. This learning algorithm starting from a *buggy* Prolog program can infer a *correct* program using unfolding transformation and the IDT debugger. Using the IMPUT method, the PECG system will be able to suggest a correct clause to the user instead of the buggy clause recognized during the debugging process.

8 Appendix

8.1 Some details on ECG representation:

- the set of attributes $\{x_l^k, y_l^k, x_m^k, y_m^k, x_r^k, y_r^k, x_{la}^k, v_{la}^k, x_{ra}^k, v_{ra}^k, A^k, E^k\}$ to define the primitive pattern peak P_k. where
  (x_l^k, y_l^k) is the left boundary of the peak P_k,
  (x_m^k, z_m^k) is the peak extremum of the peak P_k,
  (x_r^k, y_r^k) is the right boundary of the peak P_k,
  x_{la}^k is the x-coordinate of the point in the left arm of the peak P_k where the maximum (absolute) slope occurs,
  v_{la}^k is the value of the slope at the point with x-coordinate equ al to x_{la}^k,
  x_{ra}^k is the x-coordinate of the point in the right arm of the pea k P_k where the maximum (absolute) slope occurs,
  v_{ra}^k is the value of the slope at the point with x-coordinate equ al to x_{ra}^k,
  A^k is the angle at the peak extremum,
  E^k is the energy of the peak k defined as $\sim_{i=p}^{q} (y_i - y_{i-1})^2, p = x_l + 1, q = x_r$.
- the set of attributes $(x_b^k, y_b^k, x_e^k, y_e^k, B^k, C^k)$ to define the primitive pattern straight line segment k where
  (x_b^k, y_b^k) is the start point of the line segment,
  (x_e^k, y_e^k) is the end point of the line segment,
  B^k and C^k are the coefficients in the equation $y = B^k x + C^k$.

- the set of attributes $\{x_b^k, y_b^k, x_e^k, y_e^k, B^k, C^k, D^k\}$ to define the primitive pattern parabolic segment K where
  (x_b^k, y_b^k) is the start point of the parabolic segment,
  (x_e^k, y_e^k) is the end point of the parabolic segment,
  B^k, C^k, D^k are the coefficients in the equation $y = B^k x^2 + C^k x + D^k$.

8.2 The syntactic part of the logic program for ECG

```
ecg_lead     --> initial_part, cardiac_cycles, final_part.

initial_part --> segment, initial_part.
initial_part --> "".
initial_part --> peak , initial_part.

final_part   --> qrs, rest_part.
final_part   --> rest_part.

rest_part    --> segment, rest_part.
rest_part    --> peak, rest_part.
rest_part    --> "".

cardiac_cycles --> cardiac_cycle, cardiac_cycles.
cardiac_cycle  --> qrs, non_qrs.

qrs     --> q , r , s , r_ver, s_ver, r_ver2, s_ver2.
qrs     --> q , r , s , r_ver, s_ver, r_ver2.
qrs     --> r , s , r_ver, s_ver, r_ver2, s_ver2.
qrs     --> r , s , r_ver, s_ver, r_ver2.
qrs     --> q , r , s , r_ver.
qrs     --> q , r , s , r_ver
qrs     --> r , s , r_ver , s_ver.
qrs     --> r , s , r_ver.
qrs     --> q, r , s.
qrs     --> q , r.
qrs     --> r , s.
qrs     --> r .
qrs     --> qs .

non_qrs --> sr.
non_qrs --> st, t, tr.
non_qrs --> sp, p, pr.
non_qrs --> st, t, p, pr.

st      --> interwave_segment.
```

```
tp      --> interwave_segment.

pr      --> interwave_segment.

tr      --> interwave_segment.

sp      --> interwave_segment.

sr      --> segment, interwave_segment.

sr      --> peak, interwave_segment.

interwave_segment --> segment, interwave_segment.
interwave_segment --> "" ,
interwave_segment --> peak, interwave_segment.

t       --> t_or_p.

p       --> t_or_p.

t_or_p  --> "K_plus", "K_minus".
t_or_p  --> "K_minus", "K_plus".
t_or_p  --> "K_plus".
t_or_p  --> "K_minus".

r       --> "K_plus".

r_ver   --> "K_plus".

r_ver2  --> "K_plus".

q       --> "K_minus".

qs      --> "K_minus".

s       --> "K_minus".

s_ver   --> "K_minus".

s_ver2  --> "K_minus".

peak   --> "K_plus".
peak   --> "K_minus".

segment --> "E".
```

```
segment --> "Pi".
```

8.3 The name and the meaning of the arguments in the ECG program

iw	denoting the number of waves, inherited
sw	denoting the number of waves, synthesized
ic	denoting the number of classes, inherited
sc	denoting the number of classes, synthesized
$iqrs[i]$	denoting the number of QRS's in class i, inherited
$sqrs[i]$	denoting the number of QRS's in lass i, synthesized
$type$	denoting the QRS morphology, synthesized
beg	denoting the begin point of a pattern, synthesized
end	denoting the end point of a pattern, synthesized
xm	denoting the x-coordinate of the extremum of a peak, synthesized
$ybeg$	denoting the y-coordinate of the point with x-coordinate beg, synthesized
$yend$	denoting the y-coordinate of the point with x-coordinate end, synthesized
yxm	denoting the y-coordinate of the extremum of a peak, synthesized
te	denoting the energy of a peak, synthesized
ang	denoting the angle of a peak, synthesized
$ldur$	denoting the duration of the left arm of a peak, synthesized
$rdur$	denoting the duration of the right arm of a peak, synthesized
lh	denoting the height of the left arm of a peak, synthesized
rh	denoting the height of the right arm of a peak, synthesized
$tp-flag$	denoting the candidacy of a peak as a P or T (sub)pattern, inherited and synthesized.

References

1. Alexin, Z., Gyimóthy, T., Boström, H.: Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner in Proceedings of the Fifth International Workshop on Inductive Logic Programming ed. Luc de Raedt Leuven, Belgie, (1995) 437–453
2. van Bemmel J. H., Willems, J. L: Trends in Computer Processed Electrocardiograms North-Holland, Amsterdam (1976)
3. Deransart, P., Małuszyński, J.: Relating Logic Programs and Attribute Grammars Journal of Logic Programming 2, (1985) 119–156
4. 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

5. Gyimóthy, T., Paakki, J.: Static Slicing of Logic Programs in Proceedings of Automated and Algorithmic Debugging ed. Ducassé, M. Sain-Malo, France, (1995) 65–77
6. Horváth, T., Gyimóthy, T., Alexin, T., Kocsis, F.: Interactive Diagnosis and Testing Logic Programs Proceedings of the Third Symposium on Programming Languages and Software Tools, ed. Mati Tombak, Kääriku Estonia August 23-24. (1993) 34–46
7. Kókai, G., Alexin, Z., Kocsis, F.: The IDT System and its Application for Learning Prolog Programs. Proc. of the Sixth International Conference on Artificial Intelligence and Information Control Systems of Robots (AIICSR-94) Smolenice Castle Slovakia September 12-16. (1994) 315–320
8. Antti Koski: A semantic-syntactic Recognition System Based on Attribute Automata Proceedings of the Third Symposium on Programming Languages and Software Tools Kääriku, Estonia August 23-24 1993 (83-94)
9. Lloyd, J. W.: Foundations of Logic Programming, (2nd edition) Springer-Verlag (1987)
10. Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM 31:6 June (1988) 676–686
11. 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
12. 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)
13. Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)
14. Skordalakis, E.: ECG Analysis in Syntactic and Structural Pattern Recognition Theory and Applications ed. Bunke, H. and Sanfeliu, A. World Scientific (1990) 499–533
15. Willems, J. L., van Bemmel, J. H., Zywietz, C.: Computer ECG Analysis: Towards Standardization North-Holland, Amsterdam (1986)