# A Graphical User Interface for an ECG Classifier System

Jörg Nilson[1] and Gabriella Kókai[2]

[1] Department of Computer Science, Programming Languages
Friedrich-Alexander-University of Erlangen-Nürnberg
Martensstr. 3, D-91058 Erlangen, Germany
e-mail: nilson@informatik.uni-erlangen.de

[2] Institute of Informatics, József Attila University
Árpád tér 2, H-6720 Szeged, Hungary
e-mail: kokai@inf.u-szeged.hu

**Abstract.** The PECG system integrates an ECG classifier program (implemented in Prolog) with the IDT algorithmic debugger module. If the ECG classifier cannot analyze the input the built-in IDT (Interactive Diagnosis and Testing) debugger will help to find the location of the error. This paper presents a prototypical graphical user interface (GUI) of PECG and its integration with the ECG classifier program that gives the user more information about the state of the debugging process. It provides a more user-friendly environment of the system.

## 1 Introduction

In the papers [5] and [6] a system (called PECG) was presented for the classification of ECG wave-forms. This system is based on an attribute grammar approach of ECG analysis introduced by Skordalakis et al. [11].

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 [5], is to integrate Skordalakis' ECG classifier program (implemented in Prolog) with the IDTS algorithmic debugger module [4]. This integrated tool can recognize the necessity of any modification in the classifier program. If the system cannot analyze the input the built-in IDTS (Interactive Diagnosis, Testing and Slicing) debugger will help to find the false or missing clause.

Initially a simple textual user interface of PECG was given. In this paper we present a Graphical User Interface (GUI) for the ECG system.

This GUI assists the user with the identification of buggy clauses in the classifier program. By using this interface the functions of the IDTS can be easily invoked. Furthermore, the user can directly follow the debugging process by following the graphical representation of the proof tree. During the debugging the IDTS system asks the user about the correctness of the evaluated clauses. To answer these questions would be very difficult (almost impossible) without an analyzation of the graphical representation of the sub-wave-forms. The GUI of the PECG system contains a drawing tool which offers a display of an analyzation of the corresponding part of the ECG wave-form.

The PECG and the GUI parts are interacting in such a manner that the user will recognize only one process running under a homogeneous graphical user interface. One of the main programming tools was NGUIDE [7], an environment for building graphical user interfaces.

In the remainder of this paper first a brief overview of the PECG will be given, we describe the working of the system through an example, using the textual-based user interface originally provided by the system. Section 3 contains the description of the function supported by GUI, and some discussion about the implementation issues. In Section 4 the example given in Section 2.3 is repeated to introduce, step by step, the elements of the new interface in the same order as a new PECG user would see them. Finally in Section 5 a summary and comments on future work are given.

## 2 The PECG system

In this section we give a short overview of the PECG system. A more detailed description can be found in [5].
The structure of the system can be seen in Figure 1.

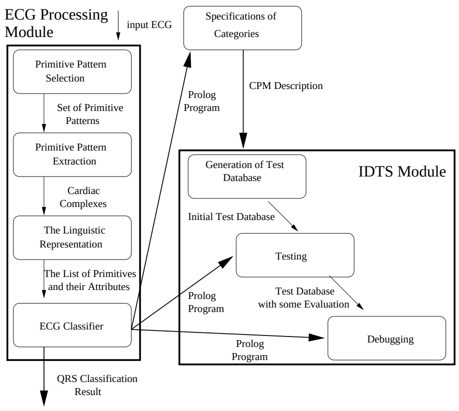


**Fig. 1.** The structure of the PECG system

### 2.1 The ECG processing module

This part of the system uses a syntactic approach to recognize an electrocardiographic pattern. The ECG processing module is built up from the following subsections:

- *primitive pattern selection:* an ECG waveform can be considered a composite pattern which can be decomposed into primitive patterns (straight line, peak, parabola) as long as it is possible.
- *primitive pattern extraction:* the main task of this subsection 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.

### 2.2 The IDTS System for Prolog programs

The IDTS system combines Shapiro's algorithmic debugging technique [10] with the CPM functional testing method [4] and with program slicing techniques [9] for making an even more advanced debugging.

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

The main idea of IDTS is the following: During the debugging of a program the user has to answer many difficult questions. If the program has already been tested, the test results of the procedures of the program can be directly applied in the debugging process without consulting the user. In addition the slicing method computes the relevant part of the proof tree.

#### 2.2.1 Shapiro's Method for Algorithmic Debugging

Shapiro's algorithms can isolate an erroneous clause, given a program and an input on which it behaves incorrectly. The algorithms traverse the proof tree of a program in different manners and ask the user about the expected behaviour of each resolved goal.

The *bottom-up* method traverses the proof tree in postorder manner and asks the *oracle* about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated with correct results the algorithm identifies the clause applied to this node as a *buggy* one. The query complexity of this method is linear in the size of the tree.

The second method investigates the nodes in *top-down* manner. If the result computed at a node is evaluated as correct by the *oracle* then the algorithm does not visit the nodes inside the sub tree. Using this approach the query complexity can be reduced to be linear in the depth of the proof tree.

The most efficient technique is the *divide and query* strategy which requires a number of queries logarithmic in the size of the proof tree. The divide and query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node in the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub tree rooted by this node. If the node's answer is correct its sub tree is removed from the tree and a new mid point is computed.

Already this description explains how important the visualization of the proof tree is.

#### 2.2.2 The CPM Testing Method

The CPM method for imperative programming languages has been defined in [8]. An extended version is presented in [3]. A formal description of the CPM for logic programs can be found in [4]. Informally the method can be described as follows: During the functional testing the programs (procedures) cannot be tested for all possible properties of their parameters. Hence, the tester's first task is to define the critical properties (*categories*) of the input parameters which are investigated in the testing process.

The categories are divided into disjoint classes — called *choices* — presuming that all the elements within a choice have an identical observable behaviour in respect of correctness. Once the categories and choices for a program have been defined all the possible *test frames* can be generated. Each test frame covers exactly one choice from each category.

#### 2.2.3 The Slicing Method

The program slicing part of IDTS is based on the *annotation* inference technique [1]. 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 define the slicing point, the *buggy* position in the program to start the slicing process.

From an annotation a *dependence graph* is constructed for the logic program [2]. A proof (refutation) tree is produced for a buggy program and using the dependence graph the tree is sliced, and the parts that have no influence on the visible symptom of a bug are removed. 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 on input arguments of the annotated program.

During the debugging the IDTS system does not investigate such a node of a proof tree which is not in the sub tree determined by the slice of the *buggy* position.

### 2.3 Small Example in PECG

In the PECG system we can choose a function from a menu. Instead of giving the description of the menu items we introduce an example and briefly explain how the system can be used to debug the description of the ECG. The example is borrowed from [5] and the syntactic part of the ECG grammar is listed in the appendix.

It is supposed that there is a bug in the logic representation of the QRS complex (the most important part of an ECG). Using the PECG version in the appendix we will see that a clause $qrs \; -- > \; r,s.$ is missing. The IDT system is used to identify this bug namely to find that an QRS class is absent in the classification. First the program generates the initial test database from the loaded CPM description. Then it lets the user try to find the erroneous clause, using Shapiro's top-down traversing method and the generated test database. The algorithm works as follows: It executes the Prolog program and at each succeeding step asks the user whether the newly determined goal is *correct* or *incorrect*.

```
Is it ok [initial_part([p_peak,par,...],[p_peak,n_peak,...])] (y/n) y
Is it ok [cardiac_cycles(0,0,[0],1,1,[1],[p_peak,n_peak,...],[])] (y/n) n
Is it ok [cardiac_cycle()] (y/n) n
Is it ok [qrs(0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,...],
         [n_peak,line,p_peak,...])] (y/n) n
Is it ok [r(253,280,275,-220,-246,461,158466,11.1,[p_peak,n_peak,line,..],
         [n_peak,line,p_peak...])] (y/n) y
```

To debug an ECG description, the user has to understand large data structures. To display and handle them a special mechanism has to be included in the GUI.

# 3 The graphical user interface for PECG: PECG-GUI

In this chapter the graphical user interface will be explained, but first a short overview is given of the New Graphical User Interface Developer's Environment (NGUIDE, [7]) which was used for programming the PECG-GUI. The explanation of the communication between the PECG and the GUI and a detailed description of the functions supported by the GUI follows.

## 3.1 Overview about NGUIDE

The main part of NGUIDE is a C++ class library, which makes it possible to apply a wide range of features for building graphical user interfaces. This class library was developed at the University of Erlangen-Nürnberg during the last two years. The main design goals of NGUIDE are:

- portability - platform independence
- efficiency - fast and easy-to-use dialogs
- object orientation - the modern way of writing programs
- extensibility - support for new types of data, new gadget types etc.
- functionality - needlessly to extend NGUIDE in most cases
- configurability - to allow even the end user to adapt the dialogs to his needs

- simplicity - to program the first dialog in a few minutes

These features, listed above, ensure that NGUIDE provides a good basis for the implementation of PECG-GUI.

In this paragraph some definitions are given, which will be used in the remaining part of the paper:

- A dialog is a window including its contents and user interaction.
- Gadgets are the atomic parts of a dialog. One gadget represents a functional or structural entity
- A graphical user interface (GUI) of a system consists of one or more dialogs on a window based display including all user interaction with this system via these dialogs. As the attribute graphical suggests, the dialogs are expected to present information in some sort of a graphical way.

### 3.2 Communication with Prolog

The GUI needs to communicate with the Prolog interpreter respectively with the PECG. This was solved by using an separate *tty* (compare [12]).

The advantage of this approach is that the textual user interface of PECG can remain almost entirely unchanged. Via the *tty* the GUI can emulate a human user working with the textual user interface of PECG.

The communication from the GUI to PECG is synchronous, but in the opposite direction the communication is asynchronous, so that the GUI will not be blocked by a possibly slow Prolog interpreter. This is important because graphical user interfaces should be able to react to user input at any time. Changing the size of a window is an example of an user input.

### 3.3 Parts of the PECG-GUI

In this section the main features of the developed GUI will be explained.

#### 3.3.1 Clickable Proof Trees

The graphically most advanced parts of the GUI are the proof trees of the debugged program. At the beginning of each debugging procedure the GUI constructs the needed one from a bracketed structure displayed by PECG. The correct graphical display is achieved by the facilities of NGUIDE. A click in the tree has to be matched with an answer to PECG.

The GUI supports the following activities:

- In each step of the debugging process the GUI labels one node of the proof tree the intended behaviour of which the debugger needs to know to go on with the debugging process.
- The user can tail the control of the bug location. In the case of the CPM method the debugging points can be selected. As well, the user interface can mark parts of the proof tree which are known to be correct or irrelevant for the problem. Likewise the user interface supports the selection of the exact location at which the slicing process is started.

- Nodes can be hidden and extended, which is a very important feature of the GUI because the ECG classifier uses a lot of parameters.

The first task is to answer *yes-no* questions. Every question represents a specific node in the proof tree of the program. But it is useful to visualize the context of the questions and the effect of the answer on the further execution of the debugging. For example: from which node is the current node produced, or which node will be evolved if the answer is *no* or *yes*.

On the image below the node in question is marked graphically. The user can answer by pressing one of the presented buttons *correct*, *incorrect* or *cancel* (see Figure 2). To help the user in the decision the currently recognized part of the ECG waveform is displayed in another window as bold outline (see Figure 3). This process can be executed repeatedly until the error is located.

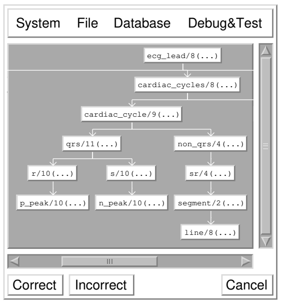


**Fig. 2.** Proof tree in the GUI - parameters are left out

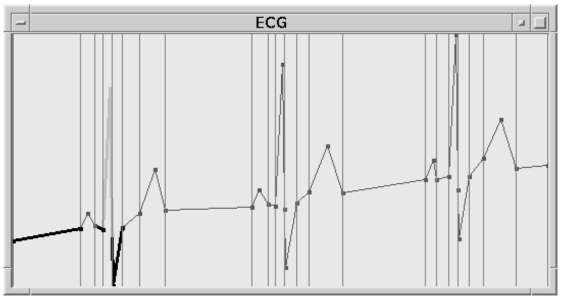


**Fig. 3.** The graphic displaying the currently asked primitive

### 3.3.2 Source Code Browsing

The only type of tree discussed so far is the proof tree which focuses on the execution of the program. But some other structures can also be displayed in tree form, for instance *static call graphs* (see Figure 4), shows which predicates could be invoked by a given procedure. With the help of this window a static program structure can be visualized as a complement to the dynamic proof tree window.

More textual displays such as sorted lists can also be useful, especially if information about textual data (e.g. Prolog predicates) should be retrieved (see Figure 4). One major factor for information retrieval is the ability to restrict the quantity of the data displayed. For instance the user may want to restrict the list of predicates displayed on the screen to predicates which begin with the letter *q* and are in the static call graph of a particular predicate. In addition the display can be restricted to predicates with a particular arity, or to predicates which match a particular name pattern, or to predicates for which facts are defined. In Figure 4 the list of predicates and a list of matching line numbers of the source code is shown.

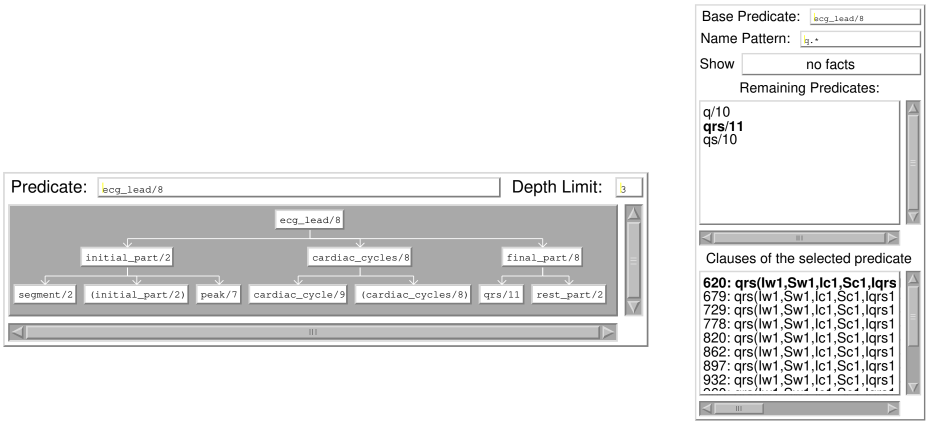


**Fig. 4. Left:** static call graph of the ECGclassifier program starting with the *ecg_lead* node, limited to a depth of 3. **Right:** information retrieval display - showing predicates beginning with the letter $q$, which occur in the static call graph displaying only the lines which contain no facts

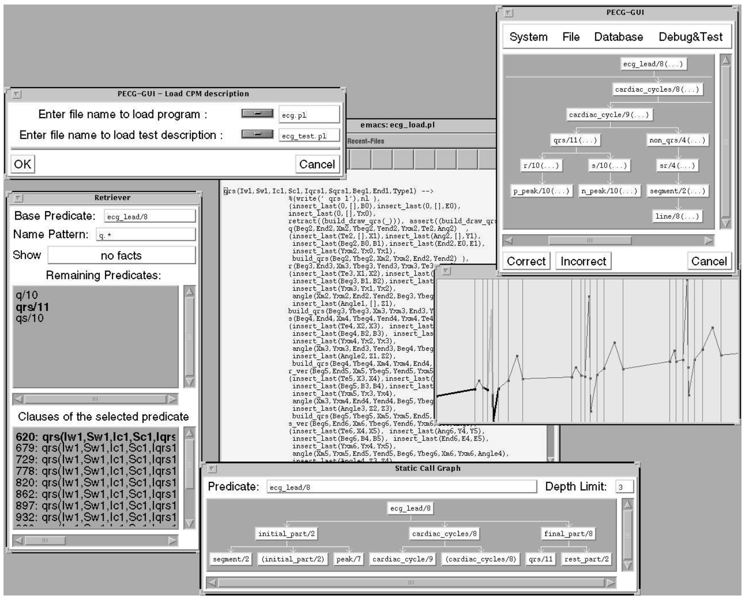


**Fig. 5.** The complete PECG-GUI with all trees, browsers and an editor

### 3.3.3 Editor Interface

By pressing a mouse button over a node of the proof tree an editor window opens and displays the part of the Prolog program which corresponds to the node, and the source code can be edited (see Figure 5). After saving the changes in the program it can be debugged with PECG again. Because *emacs* is a well known and often used text editor in the Unix environment, an interface to this editor was developed. As an alternative the editor gadget of NGUIDE could be used.

### 3.3.4 Test description and database management

The current prototype handles test descriptions in the same manner as all other input Prolog programs. There is no special handling for this type of data implemented.

Databases are used in conjunction with the whole debugging session. All data which belongs to a particular debugging session can be stored and loaded with a few mouse clicks. This data includes the name of the debugged Prolog program, the name of the description file, the name of the current test database (automatically derived from the debug session name), the current traversing method (bottom-up, top-down, divide-and-query) and the current debugging method (original Shapiro, CPM, slicing and all three together).

### 3.3.5 Integration of all components

All parts of the PECG-GUI are integrated into one process system consisting of two operating system processes, the Prolog interpreter, and the GUI. Both parts will interact in such a manner that the user will recognize only one process running under a homogeneous graphical user interface.

# 4 An example session with PECG-GUI

In this section a description of the use of PECG-GUI is presented. There is a step by step introduction in the same manner as a new PECG-GUI user would see it. During the following section the program introduced in chapter 2.3 is used.

## 4.1 Loading the program and its test description

After starting the GUI from a shell, the main window with the menu appears. Now the user has to click on the item *File* and then on the sub item *Load*. A dialog appears for entering the filename of the program and the filename of the CPM description of the program the user wants to debug. The user enters the program filename *ecg.pl* and the CPM description filename *ecg_test.pl*. Various features for easy navigation throug the file system are available. The dialog is finished by pressing the *OK* button.

After selecting the menu item *Database/Generate Database* the PECG is ready for debugging. It is assumed the GUI is already in a state that *top-down* traversing and debugging with Shapiro alone are the current settings.

### 4.2 Searching the bug

Then the menu item *Debug&Test/Test* has to be selected. The user enters the predicate `ecg_lead(e).` where file *e* contains the grammatical representation of the ECG waveform and presses the *Enter* key. Now the user is asked, whether

```
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 correct. Of course[3], the answer is *no*.

To debug the program the user selects the *Debug&Test/Debug* menu item. The predicate entered last will be shown. The user acknowledges it by pressing *Enter*. The debugging process is started. The lower part of the main window is activated and a derivation tree appears within the scrollable area. The buttons at the bottom of the dialog can now be clicked on. A node of the tree is highlighted, the user is asked whether this node is correct or not. The user answers by clicking on one of the buttons. This procedure goes on until the user cancels it, or the bug is found.

### 4.3 Fixing the bug and test again

If the bug has been found automatically an editor window is opened showing the incorrect clause (`qrs --> r.`). The user corrects the bug by introducing a new clause (`qrs --> r, s.`). After saving the program the user selects again the menu item *Debug&Test*. He presses *Enter*, because the old test predicate is still in the input field. But now the question of whether the result of the call `ecg_lead(e)`

```
NUMBER OF CLASSES =  1
CLASS [1]: NUMBER OF QRS  = 3
QRS 1  BEGIN = 253   END = 280     MORPHOLOGY: rs
QRS 2  BEGIN = 737   END = 764     MORPHOLOGY: rs
QRS 3  BEGIN = 1224  END = 1250    MORPHOLOGY: rs
```

is correct will be answered by *yes*. Now he can be sure that the bug has been fixed. Then the user may select the menu item *System/Quit*.

## 5 Conclusion, Further Work

In this paper the extension of the PECG system with a graphical user interface, the PECG-GUI has been presented. This integrated system can classify ECG waveforms. The classifier program is based on an attribute grammar description of ECG published by Skordalakis et al [11]. The system contains the IDTS

[3] Keep in mind that the user is a medician, who knows that there has to be a RS complex and not only three QRS complexes

algorithmic debugger module which can recognize if any change in the classifier program is necessary.

The IDT module improves Shapiro's original algorithmic debugging method by reducing the number of questions to the user. It combines the Category Partition Testing the program slicing with the algorithmic debugging techniques. The graphical user interface provides additional information that makes it easier for the user to answer the questions and to control the debugging process. We consider a user-friendly graphical interface to be of great importance for (successfully) using the PECG system.

Several implementation details could be improved. We are planning a special support for test description files (consistency checks, more intuitive way of definition).

At present the communication simulates a novice user of IDTS. It can be reworked to simulate an expert user i.e. calling the predicates of IDTS directly.

## References


1. Boye J., Paakki J., Maluszyński J.: Synthesis of Directionality Information for Functional Logic Programs. In: Proc. 3rd Int. Workshop on Static Analysis, Padova, 1993. LNCS 724, Springer-Verlag, 1993, 165-177
2. Deransart P., Maluszyński J.: Relating Logic Programs and Attribute Grammars. Journal of Logic Programming 2, 1985, 119-156.
3. 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
4. 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
5. 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
6. Kókai, G., Alexin, Z., Gyimóthy, T.: Analyzing and Learning ECG Waveforms, Proc. of The Sixth International Workshop on Inductive Logic Programming (ILP'96)Stockholm, Sweden 28-30 August, 1996 152-171
7. Nilson J.: NGUIDE Programmierhandbuch. Department of Computer Science – Programming Languages, University Erlangen Nürnberg, 1997.
8. Ostrand, T. J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests CACM 31:6 June (1988) 676–686
9. 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
10. Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)
11. Skordalakis, E.: ECG Analysis in Syntactic and Structural Pattern Recognition Theory and Applications ed. Bunke, H. and Sanfeliu, A. World Scientific (1990) 499–533

12. Stevens, W. R.: Advanced programming in the UNIX environment Addison Wesley (1994)
13. Westman, R., Frirzson, P.: Graphical User Interfaces for Algorithmic Debugging Proc. of the First International Workshop, AADEBUG'93 LNCS 749, Linköping, Sweden, May 1993. 273-286

## 6 Appendix

```
ecg_lead     --> initial_part, cardiac_cycles, final_part.
initial_part --> segment, initial_part. | "". | peak , initial_part.
final_part   --> qrs, rest_part. |  rest_part. | "".
rest_part    --> segment, rest_part. |  peak, 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. % the missing clause
qrs     --> r .
qrs     --> qs .
non_qrs --> sr. | st, t, tr. | sp, p, pr. | st, t, p, pr.
st      --> interwave_segment.
tp      --> interwave_segment.
pr      --> interwave_segment.
tr      --> interwave_segment.
sp      --> interwave_segment.
sr      --> segment, interwave_segment. | peak, interwave_segment.
interwave_segment --> segment, interwave_segment. | "".|  peak, interwave_segment.
t       --> t_or_p.
p       --> t_or_p.
t_or_p  --> "p_peak", "n_peak". | "n_peak", "p_peak". | "p_peak". | "n_peak".
r       --> "p_peak".
r_ver   --> "p_peak".
r_ver2  --> "p_peak".
q       --> "n_peak".
qs      --> "n_peak".
s       --> "n_peak".
s_ver   --> "n_peak".
s_ver2  --> "n_peak".
peak    --> "p_peak". | "n_peak".
segment --> "line". | "par".
```

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