Computer Methods and Programs in Biomedicine, 35 (1991) 93–110

COMMET 01194

Section I: Methodology

DTL: a language to assist cardiologists in improving classification algorithms

J.A. Kors, D.M. Kamp, D.P. Snoeck Henkemans and J.H. van Bemmel

Department of Medical Informatics, Faculty of Medicine and Health Sciences, Erasmus University, Rotterdam, The Netherlands

Heuristic classifiers, e.g., for diagnostic classification of the electrocardiogram, can be very complex. The development and refinement of such classifiers is cumbersome and time-consuming. Generally, it requires a computer expert to implement the cardiologist's diagnostic reasoning into computer language. The average cardiologist, however, is not able to verify whether his intentions have been properly realized and perform as he hoped for. But also for the initiated, it often remains obscure how a particular result was reached by a complex classification program. An environment is presented which solves these problems. The environment consists of a language, DTL (Decision Tree Language), that allows cardiologists to express their classification algorithms in a way that is familiar to them, and an interpreter and translator for that language. The considerations in the design of DTL are described and the structure and capabilities of the interpreter and translator are discussed.

Programming-language design; ECG classification; Interpreter; DTL

1. Introduction

The prevalent approach to computerized interpretation of the electrocardiogram (ECG) is to simulate the cardiologist's reasoning in classifying an ECG by means of decision trees [1]. Generally, these classifiers are complex. This complexity stems from the fact that many disease categories have to be distinguished; the decision nodes may contain comprehensive tests; and procedural knowledge is incorporated in the trees.

The development and refinement of such classifiers is an elusive task. Generally, it requires a computer expert to translate the cardiologist's diagnostic reasoning into computer language. The resulting program, however, is a sealed book to the average cardiologist, who thus cannot verify whether justice has been done to his intentions, or perhaps whether his intentions would need adjustment. But even for the expert, the workings of a complex classification program may be hidden from insight and how a particular diagnostic statement was reached may remain obscure.

In this article we describe DTL (Decision Tree Language), a simple, easy-to-learn language that can be used by domain experts for expressing their classification algorithms. Furthermore, we describe a DTL interpreter and translator. These tools allow the user to develop, test and modify the classification algorithms in an interactive way, and to generate an efficient run-time version.

The outline of the article is as follows. Firstly, we describe the system requirements and motivate the approach that was taken. Secondly, we discuss language-design criteria and their influence on the design of DTL. A description of the

Correspondence: J.A. Kors, Department of Medical Informatics, Faculty of Medicine and Health Sciences, Erasmus University, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands.

syntax and semantics of DTL can be found in the Appendices. Thirdly, the DTL interpreter and translator will be described.

2. Requirements

Four requirements for the language and its environment were discerned when we started this project:

Readability. The language should provide optimal readability for the intended users, i.e., cardiologists. Language constructs had to be available that allow cardiologists to express their knowledge in a way familiar to them. Decision trees are the main vehicle to represent such knowledge. Of particular importance is the specification of decision criteria. They involve amplitude and time-interval measurements in the so-called ECG leads. ECG leads are signals recorded by means of electrodes on the body surface, the standard ECG consisting of twelve separate leads. Cardiologists often specify criteria that involve the same type of measurement in a set of leads. They may also want to indicate that certain diagnostic categories are inhibited by the presence of others, or that a particular algorithm needs to be executed for a set of leads. The language constructs that we selected are based on material in user manuals for electrocardiographs [2,3], previous attempts in this field [4,5], and our own experience.

Interaction. It should be possible to follow the diagnostic classification of a particular patient and to modify and test the classification program on-line. This requirement suggests the use of an interpreter with at least the following capabilities: a verbose mode, the setting of breakpoints, a step mode, on-line modification, and logging facilities. The interpreter may be helpful during the implementation of a classification algorithm; its main value, however, is in the help it provides to cure diagnostic errors, inconsistencies and omissions in a classification program that is functioning properly from a programming point of view, but still can be improved from a diagnostic point of view.

Compilation. It should be possible to generate a compiled version of the program. Classification algorithms are generally evaluated by considering their performance on a large database of ECGs. A fast run-time version of the program is necessary to process such a database in an acceptable amount of time. The program must also be compilable because it had to be implemented on an electrocardiograph.

Open environment. It should be possible to incorporate routines that perform dedicated operations, such as complicated computations or I/O. The intricacies of such operations need not bother a cardiologist. The system should operate in a UNIX environment because other software that we use in this field runs under this operating system.

Several existing languages or expert system shells provide capabilities that fulfill part of the above requirements. In our opinion, none is able to fulfill all of them. The readability requirement is probably the most strict one. Although the choice of the language constructs that we deemed necessary is admittedly one out of several, we considered the constructs which are available in existing computer languages insufficient to meet our requirement for readability.

We therefore chose to design a new language and to build an interpreter for it. In order to generate an efficient run-time program, we decided that a translator that converts the language into a compiled computer language would suffice.

3. Language

There exists extensive literature on the design of programming languages (e.g., [6–8]). However, guidelines for language design are hard to formalize and may be overlapping or conflicting; programming-language design still seems to be more of an art than a science. From the above-mentioned literature we distilled three criteria that appeared to be relevant in the design of DTL: readability, reliability and compilability. We do not intend to give a comprehensive description of DTL at this place; instead we will describe several language features to illustrate how the above design criteria affected DTL. The full DTL syn-

tax is presented in Appendix A; a semantic description is given in Appendix B.

3.1. Readability

Many decisions in the design of DTL were influenced by the readability constraint. Examples will be given of DTL's appearance, data types, expressions, and control constructs.

Appearance. The meaning of language symbols should be easily recognizable. Some characteristics which affect the appearance of DTL are: (i) Identifiers consist of letters, digits, underscores, and primes. They can be of arbitrary length and all characters are significant; (ii) The assignment operator is '←' (left-pointing arrow). The '='-operator tests for equality in Boolean expressions; (iii) Comments start with '##' and extend to the end of the line; (iv) White space can be used freely to clarify the structure of the program.

Data types. (i) DTL has only four built-in data types: 'integer', 'real', 'boolean' and 'string'. These built-in types can be used to construct arrays and records. No pointers are available; (ii) Symbolic constants, including array constants, can be declared; (iii) Two special record types have been predefined: 'lead' and 'location'. The 12 standard ECG leads are available as lead variables: I, II, III, aVR, aVL, aVF, V1, V2, V3, V4, V5, V6. The 'lead' data type is used for storing lead-dependent measurements that are relevant for a classification, e.g., the duration of a Q wave, the amplitude of an R wave, the presence of a QS pattern, etc. (see Fig. 1a). A similar data type 'location' is supplied, which is used primarily in the classification of infarctions.

Expressions. (i) DTL supports ternary comparison operators; (ii) Operators distinguish only five precedence levels; (iii) Two special kinds of expression are available: lead conditions and location conditions. The general syntax of a lead condition is:

(⟨expression⟩) in [⟨number⟩ of] ⟨lead list⟩

The '⟨expression⟩' can be an arbitrary DTL condition in which lead-dependent measurements can

```
a.  types
        lead: record
              Q_dur: integer;
              R_amp: integer;
              QS_pattern: boolean;
              QR_ratio: real;
              .
              .
        end_record;
    end_types;

    variables
        I: lead;
        II: lead;
        .
        .
        V6: lead;
    end_variables;

b.  (R_amp in V3 < R_amp - 50) in 1 of V1, V2

    (QRS_pos_amp - QRS_neg_amp > 1000) in I, II, (V1..V6)

    (25 <= Q_dur < 35 and 1/4 <= QR_ratio < 1/3) in 2 of V3, V4, V5

c.  for lead in I, II, (V1..V6) do
        if (Q_dur in lead > 100) then
            .
        end_if;
        if (lead is II) then
            .
        end_if;
    end_for;
```

Fig. 1. Examples of the DTL syntax: (a) built-in lead variables; (b) lead conditions; (c) special for-loop.

occur. The optional '⟨number⟩ of' clause indicates the number of lead variables for which '⟨expression⟩' should at least be true in order for the lead condition to evaluate to true. If this clause is omitted, '⟨expression⟩' must be true in all leads specified in '⟨lead list⟩' (see Fig. 1b).

Control constructs. (i) DTL has a powerful set of control structures: 'if-then(-else)', 'for-in-do', 'loop-while', and 'loop-until'. These structures are self-bracketing, i.e., keywords terminate each construct. No 'goto'-statement is provided; (ii) A special 'for-in-do' is available which takes lead or location variables as its index variable (see Fig. 1c); (iii) Procedures and functions (program units) are supported. Parameters may be passed both 'by value' and 'by reference'. In the latter case, they have to be explicitly marked as such in the call.

The example in Fig. 2 illustrates some of the above features. A (simple) rule for classifying anterior infarction is presented. The corresponding implementation in FORTRAN, the language in which our ECG classification program was written originally, demonstrates the difference in readability.

3.2. Reliability

Several properties of DTL increase its reliability.

No default declaration. All variables must be explicitly declared before being used. The only exception is the index variable of a lead or location for-loop. This variable is declared by DTL upon entering the loop and deleted afterwards. Typing errors in the index variable will still be detected because either an undeclared variable will be referenced or an already declared variable will be

```
procedure anterior_infarction()

##  Location variable ANT and lead variables V2-5 are defined globally.

if ( (Q_dur >= 40) in 1 of V3, V4 or
     (Q_dur >= 35 and QR_ratio >= 1/3) in 2 of (V2..V5)) then
   INFARCT in ANT <- DEFINITE;
else
   if ((25 < Q_dur < 35 and QR_ratio >= 1/3) in 2 of (V2..V5)) then
      INFARCT in ANT <- PROBABLE;
   end_if;
    .
    .
end_if;
end_procedure;
```

```
      SUBROUTINE ANTINF()
      INTEGER I, COUNT1, COUNT2
C  Variables QDUR, QRRAT, INF and constants V2-5, ANT, DEF, PROB
C  are assumed to be appropriately declared.
      COUNT1 = 0
      DO I = V3, V4
         IF (QDUR(I) .GE. 40) COUNT1 = COUNT1 + 1
      ENDDO
      COUNT2 = 0
      DO I = V2, V5
         IF (QDUR(I) .GE. 35 .AND. QRRAT(I) .GE. 1.0/3.0)
     1      COUNT2 = COUNT2 + 1
      ENDDO
      IF (COUNT1 .GE. 1 .OR. COUNT2 .GE. 2) THEN
         INF(ANT) = DEF
      ELSE
      COUNT1 = 0
      DO I = V2, V5
         IF (QDUR(I) .GT. 25 .AND. QDUR(I) .LT. 35 .AND.
     1       QRRAT(I) .GE. 1.0/3.0) COUNT1 = COUNT1 + 1
      ENDDO
      IF (COUNT1 .GE. 1) INF(ANT) = PROB
      ENDIF
         .
         .
      RETURN
      END
```

Fig. 2. Example of an algorithm to classify anterior infarction implemented in DTL (upper part) and in FORTRAN (lower part).

redeclared. In both cases an error message will be given.

No default initialization. Variables and fields of arrays or record variables which have been declared but have not been assigned a value, have a special value 'undefined'. The function `undefined` is the only function that can accept a variable with the value 'undefined'. If so, it returns `true`, otherwise `false`. Using undefined variables in numerical or string expressions will cause an error message.

Type checking. DTL has strict rules for the combination of values of different types: Only `integer` and `real` types may be combined. This enables the interpreter and the translator to detect illegal combinations of values prior to program execution.

3.3. Compilability

DTL contains several language features that ease compilation.

Static types. All variables and constants must be declared. Their types cannot change during program execution. Thus, all information about operand types is available at compile time, which facilitates code generation.

Static arrays. The size of arrays has to be specified by a constant expression and cannot change during program execution. This facilitates runtime memory management and reduces the translator's complexity.

Scope rules. DTL has simple scope rules which facilitate compilation. Variables that are declared in procedures or functions are only accessible from within these units (local variables). All other variables, which must have been declared in the so-called declaration section of the program, can be accessed throughout the program (global variables). If a variable is global and a local variable with the same name is declared inside a certain unit, the name can only be used within that unit to access the local variable.

4. Interpreter and translator

In this section the DTL interpreter and translator will be described. We will concentrate on the functional aspects of the system; for a more technical description we refer to [9,10].

4.1. Interpreter

The interpreter consists of several functional units [11]: the command module, the parser, the pre-run module, and the core interpreter. The command module provides the user interface, processes commands, handles errors, and initializes and deletes data structures. The parser creates an internal representation of the DTL constructs in the form of parse trees and handles context-dependent errors, e.g., unbalanced parentheses, missing statement terminators, etc. The pre-run module stores DTL objects (constants, variables, procedures, etc.) in symbol tables. It also checks for context-dependent errors, e.g., illegal array ranges or variables that are declared more than once. The core interpreter actually executes the program. The DTL core interpreter is a 'linearizing' interpreter [11,12]. This means that the interpreter takes a parse tree (representing, for instance, a procedure call or a control statement) and breaks it down into simple instructions which are pushed on a control stack. Subsequently, the interpreter executes the instructions on the control stack; values are stored and manipulated on a value stack.

4.1.1. Commands

We chose to have a separate command language and programming language. The other option, to make all commands part of the programming language, would have complicated the core interpreter considerably because of the large number of debugging commands. However, DTL expressions and statements may be entered, prefixed by `p(rint)` and `s(tatement)` commands, respectively. Other commands that can be issued are:

- `b(reak)p(oint) <unit name> <line number>`. This command sets or removes a breakpoint on the specified line of the specified program unit. When the interpreter is executing a program, it checks every statement to see whether a breakpoint is set on one of the lines it occupies. If so, the interpreter does not (yet) execute the

statement, but echoes it and enters 'break mode'. At that point the user regains control and can issue other commands.

- `c(ontinue)`. With this command the user can resume a program that has entered break mode.
- `clear_everything`. This command deletes all data structures that were created during the current session. After this command has been issued, the interpreter is in its initial state.
- `describe <object_spec> [<output_file>]`. This command describes the specified DTL object(s). The optional `<output_file>` specifies the name of the file to which the description(s) should be written. Default is standard output.
- `help`. This command prints a short summary of the commands that the interpreter accepts.
- `load <DTL_file>`. This command loads the DTL object that is defined in the specified file, e.g., a program unit, into the interpreter's memory.
- `quit`. This command terminates the current session.
- `reset`. When an error occurs inside a unit or if break mode is entered inside a unit, the interpreter stays in that unit's environment. This means that the local variables of this unit are visible and may hide global variables of the same name. In order to get back to the 'top level' the `reset` command can be used.
- `run <batch_file>`. This command sequentially executes the interpreter commands in the specified batch file.
- `sh <shell_command>`. This command is the shell escape. The rest of the line is passed to the shell.
- `step`. This command toggles the step mode. When the interpreter is in step mode, the program is executed one statement at a time. After every statement, the interpreter enters break mode.
- `verbose`. This command toggles the verbose mode. When the interpreter is in verbose mode, every statement is shown to the user before it is executed. If a conditional statement is executed, the truth value of the condition and its possible constituents are displayed.
- `warn`. This command toggles the warn mode. When the interpreter is in warn mode, loading the definition of an object that was already defined is not possible and will result in a warning. If the warn mode is reset, the interpreter will overwrite the existing object with the newly loaded one.

The user interface has been kept small and simple. It uses the standard C input/output routines and is line-oriented.

4.1.2. Special features

The DTL interpreter has several special features: facilities for debugging the interpreter and for logging, and the inclusion of foreign units.

Debugging the interpreter. There are two interpreter commands to show the contents of the control stack and the value stack: `show_C` and `show_V`, respectively. They can be used to gain insight into the operation of the interpreter and are useful when the DTL language is to be modified or extended.

Logging. To increase insight in the classification process, it may be important to know how often a variable or constant is referenced. Therefore, every variable and constant has a log-flag attached to it. The user may toggle these flags by issuing a 'log'-command. When the log-flag of a variable is set, the interpreter counts all reads and writes performed on it. When a constant's log-flag is set, all reads are counted. If the log-flag is reset, the reference counts are not incremented. The value of the counts can be shown by means of the `describe` command. The `reset_refcnt` command resets the read or write reference count.

Logging is conditional on the value returned by a function `refcnt_permission`. This function has to return `true` for the logging to be performed. In the present implementation, it is a dummy function which always returns `true`. It may be adjusted by the user, however, to provide side-effects or additional criteria for logging.

Foreign units. Foreign units are procedures or functions written in a compiled language (C or C + +) to be called from DTL programs using the normal calling conventions. The reasons for providing them are that (i) foreign units enable the execution of computer-intensive computa-

tions in a much more efficient way than would be possible in interpreted DTL, and (ii) they allow for the incorporation of complicated algorithms, e.g., dedicated I/O routines, without having to provide a large set of DTL statements for implementing these algorithms. Thus, foreign units provide efficiency and extensibility to DTL while retaining its readability and simplicity. Foreign units are linked with the interpreter or translator. They have access to all variables and constants of the DTL program.

Fig. 3 illustrates our present use of the DTL interpreter. The ECGs that we use in our research have been processed by the signal analysis part of our ECG computer program [13]. For each ECG the measurements used by the classification program have been stored in a file. These measurements can be retrieved with foreign unit `read_ecg`, obviating a re-analysis of the ECG every time a classification is made. This setup is for research purposes only; eventually a (compiled) version of the classification program is integrated with the signal analysis program.

4.2. Translator

The translator converts DTL programs to C or C + + programs which can be subsequently compiled into executables. Its construction was relatively straightforward, since several parts of the DTL interpreter (parser, data structures) could be used for the translator as well. The translator does not provide extensive error checking, as programs are expected to have been debugged with the interpreter.

Interpretation and compilation of DTL programs may differ in two respects. First, the translator expects one main procedure which calls

```
$ interpreter
Welcome to the DTL command interpreter
>run load_ecg
    The ECG classification programs are loaded; file load_ecg contains DTL
    load statements for all DTL procedures.
>s read_ecg(10)
    The ECG measurements of a patient (no. 10) are read from a file by
    foreign unit read_ecg.
>s class_ecg()
ECG classification for patient 10
---------------------------------
Probable anterior infarction
Definite LVH (Left Ventricular Hypertrophy)
    DTL procedure class_ecg calls the DTL classification procedures. The
    classification is printed by a foreign unit called by class_ecg.
>verbose
verbose is ON
    The user wants to know why the classification 'Probable anterior
    infarction' is made and turns on the verbose mode.
>s anterior_infarction()
[  1]    if ( (Q_dur >= 40) in 1 of V3, V4 or
[  2]         (Q_dur >= 35 and QR_ratio >= 1/3) in 2 of (V2..V5)) then
false or false = false
***** CONDITION IS false
[  5]      if ((25 < Q_dur < 35 and QR_ratio >= 1/3) in 2 of (V2..V5)) then
***** CONDITION IS true
[  6]           INFARCT in AS <- PROBABLE;
    The procedure for anterior infarction is executed (cf. Fig. 2). Line
    numbers are indicated on the left. The first condition is false, the
    second is true. Truth values for subconditions are also indicated.
>sh vi antinf.tree
    The user concludes that the procedure for anterior infarction is flawed
    and enters the standard text editor to modify the file antinf.tree
    which contains the procedure.
>load antinf.tree
    Upon return, the corrected procedure is loaded and may be tested once
    again.
    .
    .
>quit
Do you really want to quit (y/n) ?
y
$
```

Fig. 3. Example of the use of the DTL interpreter. System output is bold and explanatory notes are underlined.

other procedures and functions. In DTL, this procedure starts with the keyword `main`. The main procedure is also accepted by the interpreter, of course, but it is not required since any procedure or function may be invoked directly by the user during an interpreter session.

Second, differences may exist between foreign units used by the interpreter and those used by the translator. This is caused by the fact that the interpreter's foreign units use DTL variables which are stored in the interpreter's symbol tables, while the translator's foreign units use their own (C + +) variables.

Both the interpreter and the translator have been implemented in C + + [14]. We used the Glockenspiel C + + compiler [15], operating on HP-9000 workstations.

5. Discussion

We developed the DTL environment to enable domain experts to express their classification knowledge in a familiar but algorithmic way and to provide insight into the possibly complex classification process. Our particular interest was to improve the diagnostic classification part of our ECG computer program. In our experience, the cardiologist's way of representing knowledge is closely resembled by the language DTL. We found that diagnostic classification algorithms could be translated into DTL in a natural way and immediately be understood by the cardiologist. The possibility to use dedicated routines written in a compiled computer language within DTL programs proved to be very useful. The interactive character of the system facilitated experimentation. New ideas were quickly implemented and tested, showing their viability. The cardiologist himself, sitting behind the computer, could follow the program's execution and pinpoint flaws immediately [16].

In the past, other languages have been developed to facilitate the development of heuristic ECG classifiers: DCDL (Diagnostic Criteria Decision Language) [5] and Hewlett Packard's ECL (ECG Criteria Language) [4]. There are, however, many differences between DTL and these languages. For instance, they do not support if-then-else, loop-constructs and array variables; they permit only a restricted form of program modularization; no foreign units can be used. Furthermore, the programs are compiled instead of interpreted; no particular facilities are provided to acquire insight into the actual classification process.

DTL is a concise language. Its capabilities may be extended, though, by defining new foreign units. Their use has some disadvantages: Foreign units have to be incorporated into the interpreter or translator, and differences exist between foreign units for the interpreter and for the translator. We feel these drawbacks are by far outweighted by the increased flexibility. Besides, foreign units probably will not be changed frequently, and the differences between the translator and interpreter versions are small.

Several useful extensions of the interpreter are possible. A history mechanism which allows the user to recall and edit previously entered commands would increase user-friendliness. Another extension deals with error checking. Presently, most error checking is done at run-time. Errors that exist in infrequently used parts of a program are very hard to detect. More extensive error checking by the pre-run module prior to interpretation may remedy this problem. The fact that DTL is strongly typed facilitates adding static error checking.

DTL contains several constructs dedicated to computerized ECG classification. In our opinion, however, the language is general enough to be used in other application areas. It is easy to learn, probably also for domain experts without programming experience. Especially those areas where insight in complex (heuristic) classifiers and on-line experimentation is important, may benefit from the capabilities of the interpreter.

6. Availability

Program source codes and executables are available from the authors.

Appendix A – DTL-syntax description

A modified version of Backus-Naur Form (BNF) is used. Non-terminal symbols are represented by strings consisting of lower-case letters and underscores, enclosed in ⟨ and ⟩. Terminal symbols represent themselves, but may be enclosed in single quotes (') to prevent ambiguity. The symbol → separates the left- and right-hand sides of a production. The vertical bar (|) separates alternative right-hand sides of a production. Square brackets ([ ]) enclose items that may appear once or may be omitted (optional items). Braces ({ }) enclose items that may be omitted or that may appear one or more times. The rest of this Appendix is presented on the following two pages.

Appendix B – DTL-semantics description

1. Basic definitions

1.1. White space

White space is a sequence of spaces, new lines, and tabs. It is never explicitly mentioned in the syntax description of Appendix A, but it can be placed before and after any of the non-terminal symbols in the productions.

1.2. Identifiers

An identifier consists of a letter followed by a (possibly empty) string consisting of letters, digits, underscores (_) and primes ('). Identifiers can be of arbitrary length and all characters of an identifier are significant. The underscores and the case of the letters are also significant.

Identifiers are used to name the entities in a program. Some identifiers have a predefined meaning and may not be re-used. These identifiers are called keywords or reserved words. The full list of DTL keywords is given in Appendix C.

1.3. Literals

Literals are constants that specify their own value. Literals can be numbers, truth values or strings. Integer literals and real literals are numeric literals that denote numbers. Integer literals are expressed in decimal notation and consist of one or more digits. Integers do not contain a decimal point. Real literals are also in decimal notation and consist of one or more digits with an optional decimal point, followed by an optional exponent part that consists of an e or E, followed by an integer literal that is optionally preceded by a minus sign. A real literal must contain an exponent part or a decimal point, or both, otherwise it cannot be distinguished from an integer literal.

The class of Boolean literals contains only two elements, the reserved words `true` and `false`. These literals denote truth values following the rules of Boole's algebra. String literals are arbitrary strings of printable characters, enclosed in double quotes. If a string literal is to contain a double quote, this double quote has to be preceded by another one.

1.4. Comment conventions

Comments start with '##' and extend to the end of the line. They can contain arbitrary text and can be placed anywhere except within a lexical element (i.e., identifier, operator or literal).

2. Variables, constants and expressions

2.1. Declaration of variables

All variables have to be explicitly declared by the user. Variables are declared in a 'variables block'. The declaration has the form:

```
variables
  <var_name> : <type_name>;
 {<var_name> : <type_name>;}
end_variables;
```

The `<var_name>` can be an arbitrary identifier, as long as it is not a keyword and it is not declared more than once in the same variables block.

The `<type_name>` can be one of the built-in types: `integer`, `real`, `boolean` or `string`, or it can be the name of one of the types defined by the user (see 4.1).

2.2. Assignment

Values can be assigned to variables in two ways: by means of an assignment statement and

Appendix A: continued

```
<compile_unit>      ->  <type_block>
                    |   <constant_block>
                    |   <variable_block>
                    |   <procedure_def>
                    |   <function_def>
<type_block>        ->  types
                            <type_def>; {<type_def>;}
                        end_types;
<constant_block>    ->  constants
                            <constant_def>; {<constant_def>;}
                        end_constants;
<variable_block>    ->  variables
                            <variable_decl>; {<variable_decl>;}
                        end_variables;
<procedure_def>     ->  procedure <id> ([<formal_par>{,<formal_par>}])
                            [<variable_block>]
                            <statements>
                        end_procedure;
                    |   main
                            [<variable_block>]
                            <statements>
                        end_main;
<function_def>      ->  function <id> ([<formal_par>{,<formal_par>}])
                            returns <built_in_type>
                            [<variable_block>]
                            <statements>
                        end_function;
<type_def>          ->  <id>: array '[' <int_range> ']' of <built_in_type>
                    |   <id>: record
                                <record_field_decl>;
                                {<record_field_decl>;}
                            end_record
<constant_def>      ->  <id>: <built_in_type> <- <expression>
                    |   <id>: array <- '{' <int_const> {, <int_const>} '}'
<variable_decl>     ->  <id>: <general_type>
<formal_par>        ->  <formal_in_par> | <formal_in_out_par>
<statements>        ->  [<statement>]; {<statement>;}
<built_in_type>     ->  integer | real | boolean | string
<int_range>         ->  (<int_const> .. <int_const>)
<record_field_decl> ->  <id>: <built_in_type>
<expression>        ->  <relation>
                    |   <relation> and <expression>
                    |   <relation> or <expression>
<general_type>      ->  <id> | <built_in_type>
<int_const>         ->  <id> | <int_literal>
<formal_in_par>     ->  <id>: <built_in_type>
<formal_in_out_par> ->  & <id>: <built_in_type>
<statement>         ->  <simple_statement> | <block_statement>
<relation>          ->  <simple_expr>
                    |   <simple_expr> '<'  <simple_expr>
                    |   <simple_expr> '<=' <simple_expr>
                    |   <simple_expr> '='  <simple_expr>
                    |   <simple_expr> '>=' <simple_expr>
                    |   <simple_expr> '>'  <simple_expr>
                    |   <simple_expr> '\=' <simple_expr>
                    |   <simple_expr> '<'  <simple_expr> '<'  <simple_expr>
                    |   <simple_expr> '<'  <simple_expr> '<=' <simple_expr>
                    |   <simple_expr> '<=' <simple_expr> '<'  <simple_expr>
                    |   <simple_expr> '<=' <simple_expr> '<=' <simple_expr>
                    |   <simple_expr> '>'  <simple_expr> '>'  <simple_expr>
                    |   <simple_expr> '>'  <simple_expr> '>=' <simple_expr>
                    |   <simple_expr> '>=' <simple_expr> '>'  <simple_expr>
                    |   <simple_expr> '>=' <simple_expr> '>=' <simple_expr>
<simple_statement>  ->  <assignment> | <proc_call> | print <expression>
                    |   return <expression> | exit | stop
<block_statement>   ->  <if_block> | <for_block> | <loop_block>
<simple_expr>       ->  <term>
                    |   <term> + <simple_expr>
                    |   <term> - <simple_expr>
<assignment>        ->  <var_expr> <- <expression>
                    |   <var_expr> <- undefined
<proc_call>         ->  <proc_name> ([<actual_par> {,<actual_par>}])
<if_block>          ->  if (<expression>) then
                            <statements>
```

```
                    [else
                        <statements>]
                    end_if
<for_block>     ->  for <id> in <for_list>
                        <statements>
                    end_for
<loop_block>    ->  loop
                        <statements>
                    while (<expression>)
                        <statements>
                    end_loop
                |   loop
                        <statements>
                    until (<expression>)
                        <statements>
                    end_loop
<term>          ->  <factor> | <factor> * <factor> | <factor> / <factor>
                |   <factor> div <factor> | <factor> mod <factor>
<int_list>      ->  <int_const> | <int_range> | <int_const>, <int_list>
                |   <int_range>, <int_list>
<lead_list>     ->  <lead> | <lead_range> | <lead>, <lead_list>
                |   <lead_range>, <lead_list>
<loc_list>      ->  <loc> | <loc_range> | <loc>, <loc_list>
                |   <loc_range>, <loc_list>
<primary>       ->  <int_literal> | <real_literal>
                |   <boolean_literal> | <string_literal> | <id>
                |   undefined (<expression>) | <array_field_expr>
                |   <record_field_expr> | <function_call>
                |   <lead_condition> | <lead_equality>
                |   <loc_condition> | <loc_equality> | (<expression>)
<record_var_name> -> <id> | <lead> | <loc>
<lead_range>    ->  (<lead> .. <lead>)
<loc_range>     ->  (<loc> .. <loc>)

<function_call>     ->  <id> ([<actual_par> {, <actual_par>}])
<lead_condition>    ->  (<expression>) in [<int_const> of] <lead_list>
<lead_equality>     ->  <lead> is <id> | <id> is <lead>
<loc_condition>     ->  (<expression>) in [<int_const> of] <loc_list>
<loc_equality>      ->  <loc> is <id> | <id> is <loc>
<id>                ->  <letter> { <printable> }
<lead>              ->  I | II | III | aVR | aVL | aVF
                    |   V1 | V2 | V3 | V4 | V5 | V6
<loc>               ->  INF | HL | ANT | AS | AL | POST
<int_literal>       ->  <digit> { <digit> }
<real_literal>      ->  [<int_literal>] . <int_literal> [<exp>]
                    |   <int_literal> <exp>
<boolean_literal>   ->  true | false
<string_literal>    ->  " {<string_char>} "
<var_expr>          ->  <id> | <array_field_expr> | <record_field_expr>
                    |   (<var_expr>)
<proc_name>         ->  <id> | main
<actual_par>        ->  <actual_in_par> | <actual_in_out_par>
<for_list>          ->  <int_list> | <lead_list> | <loc_list>
<factor>            ->  <primary> | not <primary> | abs <primary>
                    |   <primary> ** <primary> | - <primary>
<array_field_expr>  ->  <id> '[' <expression> ']'
<record_field_expr>     ->   <id> in <record_var_name>
<actual_in_par>     ->  <expression>
<actual_in_out_par>     ->   & <var_expr>
<letter>            ->  a | b | c | d | e | f | g | h | i | j | k | l | m
                    |   n | o | p | q | r | s | t | u | v | w | x | y | z
                    |   A | B | C | D | E | F | G | H | I | J | K | L | M
                    |   N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<printable>         ->  <letter> | <digit> | _ | '
<digit>             ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<exp>               ->  e [-] <int_literal> | E [-] <int_literal>
<string_char>       ->  <printable> | ""
```

by means of a procedure or function call with in/out parameters. Procedure and function calls will be discussed in Section 5.

An assignment statement consists of two expressions, separated by a ← sign. The left-hand side (lhs) of the assignment statement specifies the location where the value of the other expression, the right-hand side (rhs), is to be stored.

In DTL one can only assign to simple variables (variables of a built-in type). As a consequence of this, the lhs is a restricted expression that can only be of one of the following three forms: the name of a simple variable, the specification of a field of a record variable, and the specification of a field of an array variable.

The rhs is a general expression that is only restricted in the type of values that it can have. This type must be the same as the type of the lhs or coercible to it.

2.3. *Undefined variables*

When variables are declared but no value has been assigned to them, they are initialized to a special value: 'undefined'. When undefined variables are used as part of an expression, the expression normally also becomes undefined. An exception to this rule is the way in which Boolean expressions (conditions) are handled (see 2.5.2).

Undefined expressions may not be used as rhs of assignment statements. A variable may be assigned an undefined value by using the reserved word `undefined` as the rhs of an assignment statement. It is also illegal to pass undefined variables as in-parameters to a procedure or function. The only exception to this rule is the predefined function `undefined()` which expects a general expression of a built-in type as its parameter and returns `true` if the expression is undefined.

2.4. *Symbolic constants*

Symbolic constants are declared in a 'constants block'. The declaration has the form:

```
constants
  <constant_def>;
 {<constant_def>;}
end_constants;
```

Constant definitions have the following syntax:

```
<id> : <built-in_type> ←
<expression>
```

or

```
<id> : array ←
 '{ '<int_const>{,<int_const>} '} '
```

The identifier (`<id>`) in the production is the name of the symbolic constant. This can be any identifier that is not also a global variable or the name of another symbolic constant. In the first production, the expression that is used to supply the constant's value is an unrestricted expression, that can use all operators and all symbolic constants that are known at the time of definition.

The second production is the syntax of the definition of array constants: symbolic constants that represent one-dimensional arrays of integers. The indices for an array constant range from 1 to the array's number of elements. Array constants are accessed via the same type of expression as array variables (see 4.2). Symbolic constants can appear in all types of expression, but not as the lhs of an assignment statement or as in/out parameters (see 5.3).

2.5. *Expressions*

An expression must always have a value of a built-in type. Expressions consist of literals, symbolic constants, variables, fields of record variables, fields of array variables or constants, function calls and ECG conditions, combined by means of operators and brackets. There are three types of expressions: numerical, Boolean and string.

2.5.1. Numerical expressions. Numerical expressions are used to supply the program with integer or real values. The simplest numerical expressions are the ones without operators. They can be of the following forms: Integer literal, real literal, identifier (representing a numerical variable or constant), array field (of a numerical array variable or constant), record field (a numerical field of a record variable), function call (of a function with a numerical return type).

The unary operators that are defined on numerical values are: `abs` (absolute value), and - (negation). The result of both operators is of the same type as the operand. The binary operators

that are defined on numerical values are: + (addition), - (subtraction), * (multiplication), / (division), ** (exponentiation), div (integer division), and mod (modulus). The result of +, -, and * is an integer if both operands are integers; otherwise it is a real. The result of / and ** is always a real. Both operands and result of div and mod are integers. All numerical operators return the value 'undefined' if one of their operands is undefined.

2.5.2. Boolean expressions. Boolean expressions (conditions) are used in tests that are part of conditional statements or loops. The simplest Boolean expressions are the ones without operators. They can be of the following forms: Boolean literal, identifier (representing a Boolean variable or constant), array field (of a Boolean array variable), record field (a Boolean field of a record variable), function call (of a function with a Boolean return type). There is one unary Boolean operator: not. This operator expects a Boolean operand and returns its negation. If the operand is undefined, the result is undefined as well.

There are two binary Boolean operators: and and or. These operators also expect Boolean operands and return a Boolean value. If one operand of and is undefined, and the other one is false, the operator returns false. If the other operand is true, the operator returns undefined. If one operand of or is undefined, and the other one is true, the operator returns true. If the other operand is false, the operator returns undefined. The following binary operators compare values of a built-in type and return Booleans that describe the relation between these values: = (equal), /= (not equal), > (greater than), >= (greater than or equal), <= (smaller than or equal), and < (smaller than). The operands of = and /= can be any of the built-in types provided they are of the same type. The operands of >, >=, <=, and < must be either both numerical or both strings.

DTL also supports ternary comparison operators, which can compare three numerical or string values: > >, >= >, > >=, >= >=, <= <=, < <=, <= <, and < <.

2.5.3. String expressions. String expressions are used to store or print messages. The simplest string expressions are those that are of one of the following forms: String literal, identifier (representing a string variable or constant), array field (of a string array variable), record field (a string field of a record variable). The only binary operator that is defined on strings is +, which concatenates two strings and returns the result as a new string.

2.5.4. Operator precedence. A precedence level is associated with each operator. If an expression contains two operators of different precedence levels, the one with the highest level is applied first, unless parentheses are used to force another order of evaluation. Operators of the same precedence level are left-associative. The following precedence levels are distinguished:

5: **, not, abs, (unary) -
4: div, mod, *, /,
3: +, -
2: >, >=, =, <=, <, /=
1: and, or

2.6. Coercion

Most of the numeric operators that are described in 2.5.1 and all of the comparison symbols of 2.5.2 accept combinations of reals and integers as their operands. All integers in the expression are converted to reals and then the whole expression is evaluated, using only the operators (or comparison symbols) for reals.

Another (but similar) use of coercion is when an integer literal is assigned to a real variable. The literal's integer value is first converted to a real and then the assignment statement is executed.

Every other mixing of types is illegal.

3. Control structures

3.1. If-blocks

```
if (<expression>) then
 <statements>
end_if;
```

or

```
if (<expression>) then
 <statements>
else
 <statements>
end_if;
```

The first kind of if-block checks whether <expression> (a condition) has the value 'true' and if this is the case, executes the statements after then. If the condition has the value 'false' the statements after then will be skipped. In both cases the program resumes execution after end_if.

In an if-block of the second kind, the statements after else will be ignored if the condition holds; if it does not, they will be executed.

3.2. Loop-blocks

```
loop
 <statements>
until (<expression>)
 <statements>
end_loop;
```

or

```
loop
 <statements>
while (<expression>)
 <statements>
end_loop;
```

The first loop-block executes the statements after loop, then checks whether <expression> holds. If it does not, the statements after the test are executed and execution resumes at the statement following the loop keyword. If the condition holds, control is transferred to the statements following end_loop. The second loop-block behaves almost identically to the first. The only difference is that the condition is used as a continuation condition rather than as a termination condition.

3.3. For-blocks

```
for <id> in <int_list> do
 <statements>
end_for;
```

<int_list> is a list of integers and/or integer ranges, separated by commas. The integers can be either integer literals, integer variables or integer symbolic constants. Integer ranges are of the form (<int_val1>..<int_val2>), where <int_val2> is greater than or equal to <int_val1>.

The index variable of an integer for-loop must be declared before it is used in the for-loop. Its type must be integer. The value of the index variable should not be changed by the statements in the loop. After termination of the loop, the value of the index variable is the last value in the list.

4. User-defined types

4.1. Type definition

The general format of a type definition is:

```
types
  <type_def>;
 {<type_def>;}
end_types;
```

DTL contains two user-defined types, the array type and the record type. The specification of an array type has the form:

```
<id> : array '[ '<int_range> ']' of
  <built-in_type>;
```

The specification of a record type is as follows:

```
<id> : record
  <id> : <built-in_type>;
 {<id> : <built-in_type>;}
end_record;
```

4.2. Array variables

When a new array type has been defined, variables of this type can be declared in the normal way. Variables of an array type can be accessed one field at the time. An array element can be specified by the name of the array variable followed by an integer expression denoting the index of the desired element. The integer expression must be enclosed in square brackets. Array elements can be used in all situations where normal variables of the same type can be used. This means that array elements can be used in expressions, as parameters to functions and procedures and as lhs of assignment statements. Arrays cannot be returned by functions or passed to procedures or functions. Only one-dimensional arrays are allowed.

4.3. Record variables

Record variables are declared in the same way as all other variables. After a record variable has

been declared, its fields are accessible via a record-field specification:

```
<record_field_name> in
  <record_var_name>
```

The first identifier is the name of one of the fields that are defined for records of this type. The second identifier is the name of the record variable to be accessed. Record fields can be used in all situations where normal variables of the same type can be used. This means that they can be used in expressions, as parameters to functions and procedures and as lhs of assignment statements.

Record variables can only be accessed one field at the time. Record variables cannot be returned by functions or passed to procedures or functions.

5. *Program units*

5.1. *Procedures*

The definition of a procedure has the following syntax:

```
procedure <id> (
  [[&]<id>:<built-in_type>
 {,[&]<id>:<built-in_type>}])
[<variables_block>]
<statements>
end_procedure;
```

The first `<id>` is the name of the procedure. It is followed by the list of formal parameters. The parameters can be preceded by an ampersand (&). An ampersand indicates that the parameter is an in/out-parameter, whereas no ampersand means that the parameter is an in-parameter (see 5.3). The parameter list is optionally followed by a variables block (see 2.1) that specifies the procedure's local variables. This block, in turn, is followed by the statements that perform the procedure's actions. Procedures can be called by other parts of the program by specifying the name of the procedure and a list of actual parameters that the procedure uses. The in/out-parameters must be preceded by an ampersand.

The DTL compiler demands one `main` procedure which is the program's starting point. Its syntax is:

```
main
 [<variables_block>]
 <statements>
end_main;
```

A `main` procedure need not be specified when the interpreter is used.

5.2. *Functions*

Functions are defined in the following manner:

```
function <id> (
  [[&] <id>:<built-in_type>
 {,[&] <id> : <built-in_type>}])
returns <built-in_type>
 [<variables_block>]
 <statements>
end_function;
```

The only non-trivial difference between procedure and function definitions is the clause `returns <built-in_type>`. This clause specifies the return type of the function.

The way a function returns a value is by executing the statement

```
return <expression>;
```

When this statement is encountered, execution of the function is terminated and the value of `<expression>` is returned to the calling unit. The type of `<expression>` must be the same as, or coercible to, the return type of the function. Return statements can be placed anywhere within a function (even in loops) and they can occur more than once in one function specification. If, during execution of a function, the program runs into the `end_function`, a run-time error is raised. Return statements can only occur in functions.

Function calls have the same syntax as procedure calls. Function calls can be used in general expressions, but not as lhs of assignment statements or as in/out parameters for a procedure or other function.

5.3. *In- and in/out-parameters*

Program units may be provided with two kinds of actual parameters: in- and in/out-parameters. An in-parameter is a parameter that the calling

unit uses to supply the called procedure or function with a value. An actual in-parameter can be any kind of expression whose value is of the same type as the corresponding formal parameter or coercible to it. In-parameters behave like local variables of the called unit. Their value can be used in expressions and new values can be assigned to them but these changes do not affect the caller's variables in any way.

Functions return exactly one value. Returning multiple values is done by means of the in/out-parameter mechanism. Both actual and formal in/out-parameters must be preceded by an ampersand. Unlike an actual in-parameter, which specifies a value, an actual in/out-parameter specifies a location in memory that may or may not contain a value. The called unit can use its formal in/out-parameters just like formal in-parameters or local variables. The main difference between in- and in/out-parameters is that changing a formal in/out-parameter causes the corresponding actual parameter to change as well. Another difference is that an actual in/out-parameter needs to be of the same type as its corresponding formal one, whereas for in-parameters it is enough for the actual parameter to be coercible to the formal parameter's type.

5.4. Return from a unit

There are three ways of terminating (part of) a program:

`stop:` Terminates the whole program, independent of the current position.

`exit:` Terminates the current procedure. This instruction cannot be used in functions, since it does not specify a return value.

`return:` Terminates the current function instance, specifying a return value. This instruction can only be used in functions.

5.5. Scope and visibility

Variables that are declared in the declaration section of the program, can be accessed throughout the program. These variables are called 'global'. It is also possible to declare variables in procedures or functions, in which case they are only accessible from within these units.

If a variable is global and a local variable with the same name is declared inside a certain unit, the global variable is 'invisible' within this unit.

5.7. Notes on memory management

The current implementation of the DTL interpreter uses a static memory-management scheme. This means that all variables and constants are allocated once and deallocated only when the program is terminated. A consequence of this way of managing memory is that DTL does not support recursion.

The fact that DTL uses static memory-management does not mean that the local variables in DTL are static in the FORTRAN sense, however. When a unit is exited, all its local variables are set to 'undefined' and their original values are lost.

6. ECG-specific features

DTL contains a number of features that make it especially fit for application in computerized ECG classification.

6.1. Lead variables

An ECG classification program uses a number of measurements of the ECG at hand. These measurements are time intervals or amplitudes of waves in the leads of the ECG. DTL supplies a data type, a record type named 'lead', which can be used to store these lead measurements.

The language predefines twelve lead variables of which the names are key words: `I, II, III, aVR, aVL, aVF, V1, V2, V3, V4, V5, V6`. These variables can be accessed like normal record variables. One can make lists of these lead variables and, like integer lists, these lists can contain ranges. The sequence that is defined on the lead variables is the one shown above.

6.2. Location variables

During the classification process, the program needs to record its conclusions and assumptions about the condition of the heart. Much of these data apply to specific locations of the heart. DTL

has a predefined record type named 'location' for this purpose. The following predefined variables are of type 'location': `INF, HL, ANT, AS, AL, POST`.

Like lead variables, location variables can be used in both lists and ranges.

6.3. Special for-loops

It is often necessary to execute repeatedly the same action for a number of leads or locations. For this purpose, DTL has a special for-loop construct. The syntax of such a for-loop is the same as for the normal for-loop (see 3.3), except for the iteration list which contains leads or locations instead of integers.

The index variable of a special for-loop can be used as a normal 'lead'/'location' record variable. The index variable is not a record variable, however. It can be printed, in which case it yields a small integer representing the sequence number of the current lead/location. This means that the index variable, unlike a real lead/location variable, can be used as a parameter. All other manipulations that treat the variable as an integer are illegal.

To test whether a particular lead/location in the iteration-list has been reached, DTL contains a special comparison operator: `is`. This operator can be used to compare the current value of the index variable with a particular lead/location variable. The declaration of the index variable is done automatically. The variable is declared before the program enters the for-block and deleted when the for-block is exited. Using the variable afterwards results in a run-time error. The name of the index variable cannot hide the name of a local variable; it must be given a name that has not already been given to a local variable.

As in a normal for-loop, it is illegal to assign values to the index variable or use it as an in/out-parameter.

6.5. Special conditions

A special type of condition was designed that can test multiple ECG variables at once and that conforms to the notation that is commonly used in cardiology.

These conditions can either apply to collections of lead variables or collections of location variables and are therefore called lead conditions or location conditions. The syntax of a lead condition is:

```
(<expression>) in
  [<int_const> of] <lead_list>
```

The `<expression>` can be an arbitrary DTL condition (but not a lead/location condition), in

Appendix C – DTL key words

```
abs             and             array           boolean
constants       div             do              else
end_constants   end_for         end_function    end_if
end_loop        end_main        end_procedure   end_record
end_types       end_variables   exit            false
for             function        if              in
integer         is              loop            main
mod             not             of              or
print           procedure       real            record
return          returns         stop            string
then            true            types           undefined
until           variables       while
I               II              III             aVR
aVL             aVF             V1              V2
V3              V4              V5              V6
AL              ANT             AS              INF
POST
```

which fields of the 'lead' record type can occur. The (optional) <int_const> of clause indicates the number of lead variables for which <expression> should (at least) be true in order for the lead condition to evaluate to true. If this clause is omitted, <expression> must be true in all of <lead_list>'s elements in order for the lead condition to become true. An undefined <expression> counts as false. The syntax of location conditions is similar to that of lead conditions.

Acknowledgment

We would like to thank Dr. D. Grune (Free University, Amsterdam) for helpful discussions concerning the paper.

References

[1] J.A. Kors and J.H. van Bemmel, Classification methods for computerized classification of the electrocardiogram. Methods Inform. Med. 29 (1990) 330–6.

[2] Physicians' Guide to Marquette Electronics' Resting ECG Analysis (Marquette Electronics Inc, Milwaukee 1987).

[3] Mingocare Overreading Dictionary (Siemens-Elema AB, Solna 1985).

[4] R.A. Balda, A.G. Vallance, J.M. Luszcz, F.J. Stahlin and G. Diller, ECL: A medically oriented ECG criteria language and other research tools. *in*: Ostrow HG, Ripley KL, eds. Computers in Cardiology. pp. 481–495 (IEEE Comp Soc, Long Beach: 1978).

[5] P. Rubel, P. Arnaud and D. Prevot, Système d'aide à la decision. Application à l'interpretation automatique des vectogrammes. Int. J. Bio-Med. Comput. 6 (1975) 193–211.

[6] C.A.R. Hoare, Hints on Programming Language Design. Report CS-73-403 (Stanford University, 1973).

[7] C. Ghezzi and M. Jazayeri, Programming Language Concepts (John Wiley & Sons, New York, 1982).

[8] J.P. Tremblay and P.G. Sorenson, The Theory and Practice of Compiler Writing (McGraw-Hill, New York, 1985).

[9] D.M. Kamp and D.P. Snoeck Henkemans, An Interactive Environment for Automated ECG Classification. Internal report (in Dutch). Rotterdam: Department of Medical Informatics (Erasmus University, Rotterdam, 1989).

[10] D.P. Snoeck Henkemans and D.M. Kamp, DTL User Manual (Department of Medical Informatics, Erasmus University, Rotterdam, 1989).

[11] P.J. Brown, Writing Interactive Compilers and Interpreters (J Wiley & Sons, New York, 1979).

[12] R. Bornat, Understanding and Writing Compilers. a Do-it-yourself Guide (Macmillan, London, 1979).

[13] J.A. Kors, J.L. Talmon and J.H. van Bemmel, Multilead ECG analysis, Comp. Biomed. Res. 19 (1986) 28–46.

[14] B. Stroustrup, The C++ Programming Language. (Addison-Wesley, Reading, 1987).

[15] The designer C++ Programming System, User Guide and System Manual (Glockenspiel Lim, Dublin, 1988).

[16] J.A. Kors, G. van Herpen, T. de Jong and J.H. van Bemmel, Interactive optimization of heuristic ECG classifiers (submitted for publication).