ELSEVIER SCIENTIFIC PUBLISHERS IRELAND

Computer Methods and Programs in Biomedicine 41 (1993) 107–119

computer methods and programs in biomedicine

Pre-compiling medical logic modules into C++ in building medical decision support systems

X. Gao*, B. Johansson, N. Shahsavar, K. Arkad, H. Åhlfeldt, O. Wigertz

Department of Medical Informatics, Linköping University, S-581 83 Linköping, Sweden

(Received 16 April 1993; accepted 2 August 1993)

Abstract

Development of medical knowledge bases is a time-consuming process, and no single medical institution can develop medical knowledge bases covering all areas of medicine. The use of medical knowledge representation standards such as the Arden Syntax is an attempt to enhance the writability and readability of computer-stored knowledge and facilitate transfer and sharing among institutions. A method for the realisation of decision support systems based on knowledge formulated according to the Arden Syntax is presented. An essential tool in this process is a medical logic module (MLM) pre-compiler, translating MLMs into an object-oriented programming language, C++. Advantages of the C++ approach compared with other alternatives are discussed.

Key words: Decision support; Arden Syntax; Pre-compiler; Medical logic modules

1. Introduction

Experiences from attempts to migrate decision support systems from research laboratories into clinical environments have shown that this process is far from trivial. Some of the problems encountered have involved the need for coupling between the knowledge base and existing clinical data-bases, and the need for a critical mass of knowledge. One implication of these needs is the necessity for a medical knowledge representation standard for knowledge sharing, since no individual institution can develop knowledge bases that cover all aspects of medicine. Moreover, the representation format should make the logic easier to write and to read. This would then allow medical experts, who might not be computer-literate, to provide and input their knowledge more easily and also to understand and accept what someone else has contributed [1–4].

A standard knowledge representation format for sharing medical knowledge bases among institutions has been adopted by the international standards committee of the American Society for Testing and Materials as standard E1460 [5]. The knowledge representation standard, called Arden

* Corresponding author. Tel.: +46 13 227573; Fax: +46 13 104131; E-mail: gaoxi@ami.liu.se.

0169-2607/93/$06.00
SSDI 0169-2607(93)01409-U

Syntax, is being developed jointly by groups at LDS Hospital, Salt Lake City, Utah; Linköping University Medical Informatics Department, Linköping; Erasmus University, Rotterdam; and Columbia-Presbyterian Medical Center, New York. The syntax is derived largely from the HELP system [6] and CARE, the language of the RMRS System [7].

To fulfil the ideal of sharing knowledge among different sites, issues involving implementation and execution as well as methods and tools are needed for realisation of decision support in a clinical environment, as the medical logic modules (MLMs) should be interpreted into a form that can be executed by a computer system.

In this paper we present an MLM pre-compiler for building decision support systems (DSS). The main issue addressed in this paper is how to capture the semantics of the Arden Syntax using the object-oriented programming language C++. Alternative approaches, such as expert system shell and pseudo-code, are described in the paper. Furthermore, a detailed description of the MLM pre-compiler from a software engineering point of view is reported. Finally, a prototype of a critiquing system for anti-coagulant therapy has been built using the MLMs in C++ code produced by the pre-compiler, together with an MLM engine [8].

2. The Arden Syntax

The basic concept of the Arden Syntax is that the medical knowledge is broken down into independent MLMs, each one intended to operate as a discrete module. MLMs are evoked according to events such as updates in patient data-bases. Despite its modularity, the Arden Syntax can permit an MLM to trigger the invocation of other MLMs [2,5].

The Arden Syntax aims at facilitating the sharing of computerised medical knowledge bases among personnel, information systems and institutions. There are many different types of medical knowledge, and it is impossible to accommodate everything; the scope, therefore, has been limited to such knowledge as can be represented as a set of discrete modules. Each MLM contains sufficient knowledge to make a single decision. The main focus of the MLM is to provide the generation of alerts, management suggestion and critiques, protocols, and diagnosis scores. Management information is also contained in each MLM in order to maintain knowledge bases and a link to other sources of knowledge such as medical literature and the medical entity dictionary. Physicians or other domain experts can create MLMs directly through a carefully designed MLM editor [9], or by learning the syntax. After compilation, the resulting MLMs can be used directly by a decision support system.

The maintenance category contains the slots that specify information such as title, file name, version, institution, author, specialist, date, and validation. The collected information is unrelated to the medical knowledge in the MLM and used for maintenance of the MLMs.

The slots in the library category are pertinent to knowledge base maintenance and related to the knowledge part of the MLMs. The slots, such as purpose, explanation, keywords and citations, provide medical staff with pre-defined explanatory information and a link to the medical literature.

In the knowledge category, the slots specify the functionality of the MLM. The data slot defines the terms used in the MLM, the evoke slot decides the context in which the MLM should be evoked, the logic slot tests the medical conditions before a medical decision is taken, and the action slot describes medical decision to be taken when the condition is true.

An MLM is a text file that is composed of three parts: maintenance, library, and knowledge. The information included in the MLM is used for different purposes, such as maintenance of knowledge bases, references to medical literature, and specifying the MLM task.

Experience from physicians who have used the Arden Syntax shows that the language is easy to write and read with little prior computer training [10]. An example of an MLM used in the prototype system is shown in Fig. 1.

3. The C++ interpretation approach for the Arden Syntax

There are several alternatives when implementing the Arden Syntax. In our approach, the

```
Maintenance:
Title:          Prescription without PT test;;
Filename:       pt_601;;
Version:        1.01;;
Institution:    Linköping University Hospital;;
Author:         "name";;
Specialist:     ;;
Date:           1990-06-03;;
Validation:     Testing;;
Library:
Purpose:        Give alert if there is a prescription without a test;;
Explanation:    No dose should be given if there is no new PT
                percentage;;
Keywords:       anticoagulant;;
Knowledge:
Type:           Data-driven;;
Data:
                Prothrombin_percentage:= read last
                ({PT} where it occurred within the past 12 hours);
                new_weekly_dose:= read last
                ( {WDOS} where it occurred within the past 12 hours);
                signature:= event{storage of signature};;
Evoke:          signature;;
Logic:
                if not(Prothrombin_percentage is present)
                and (new_weekly_dose  is present) then
                  conclude true;
                else
                  conclude false;
                endif;;
Action:
                write "No new dose must be given without a new PT
                percentage!";;
End:
```

Fig. 1. Example of an MLM for supervision of drug prescription.

MLMs are pre-compiled into C++ [11]. Syntax checking is performed, and evoke and trigger definition tables are produced based on the statements in the MLM source. A set of C++ modules are thus produced, which are compiled with a standard C++ compiler and linked together with an MLM engine (also implemented as a C++ program). Alternatives such as utilisation of expert system shells or development of an MLM interpreter that executes the MLMs line-by-line during run time will be discussed further on.

3.1. The C++ approach

The goals of object-oriented programming are to improve programmer productivity by increasing software extendability and reusability and to control the complexity and cost of software maintenance with expressive modularity [12]. C++, which is an object-oriented programming language, is a superset of the C language. The nature of the object-oriented language makes the compiled C++ code similar to the MLM source. The use of standard C++ compilers, debuggers and linkers is one advantage of this method for building DSSs. Errors in MLMs could actually be selected by the parser and the C++ compiler. This is an interesting problem, which, however, will be dealt with elsewhere. This approach makes translation work from MLM to C++ easier compared with alternative code generation.

The results of the pre-compilation process are

three-fold: a C++ function, an evoke definition table, and a triggger information table. An MLM engine as a run time server decides which MLMs should be executed according to the evoke definition table when receiving trigger requests from the patient data-base. Trigger information is collected so that the active data-base system is able to set up triggers in the patient data-base. The active data-base system should control the event conditions such as updates in the data-base and send the cor-

```
#include "mlm_to_c++.hxx"
mlm_type &pt_601() {
// Declaration of variable
mlm_type Prothrombin_percentage;
mlm_type new_weekly_dose;
mlm_type signature;

mlm_type NULL_TYPE;
BOOLEAN_TYPE conclude=false;
mlm_type action_message;
// data slot part

Prothrombin_percentage.read_last(1 , "PT", " it occur
within_the_past (12 hours) ");
new_weekly_dose.read_last(1 , "WDOS", " it occur
within_the_past (12 hours) ");

// evoke critiria is in <Evoke_table> file

// logic slot part
while (1) {
   if ( ! (Prothrombin_percentage.present())
   && (new_weekly_dose.present()))
   {
     conclude=true;
     break;
   }
   else
   {
     conclude=false;
     break;
   }
break;
}
// action slot part
if(conclude)
{
// do action
  action_message="No dose must be given without a new
PT percentage! (ref: pt_601)";
  write(action_message);
}
NULL_TYPE=null;
return NULL_TYPE;
}
```

Fig. 2. A C++ function generated by the MLM pre-compiler. The nature of the object-oriented language makes the compiled code similar to the MLM source, as can be seen when comparing Figs. 1 and 2. During run time, execution of the function is initiated by a trigger message from the DBMS caused by storage of a signature in the data-base. The MLM engine schedules the execution of the MLMs based on the information in the evoke definition table (see also Fig. 3).

responding triggers when the conditions become true, without user or application intervention [13].

The translated C++ code is shown in Fig. 2 (compare with the MLM source shown in Fig. 1).

3.2. The translation schema

The translation schema is separated into two parts. One is a C++ schema for data, logic and action slots and the other is a schema for evoke slots. Each module is translated into a C++ function. The evoking criteria for the module are stored in an evoke definition table, and trigger information is produced for the data-base management system.

3.2.1. The C++ schema

During pre-compilation each MLM is translated to a C++ function. Variables within one MLM are declared as objects of a given MLM class, named *mlm_type*. This class contains methods for the data operators, query aggregations and return options defined in Arden Syntax [11]. The syntax contains data types such as numeric, string, time, duration, list and Boolean, but a data type declaration is not required in the MLM source. Handling of data types during run time (when, for instance, reading data from the patient data-base) is encapsulated in the methods of the *mlm_type* class.

Since Arden Syntax allows nesting of MLMs, each compiled MLM returns an object from an MLM class. The return value is NULL when the MLMs do not have a return statement. Each call statement is translated into a call to a C++ function.

Operator overloading in C++ has been very useful. A wide meaning for operators is required, as Arden Syntax includes special data types such as time, duration and list in addition to number, Boolean and string. Operator overloading can define a special meaning for operators when applied to objects of a specific class; therefore MLM expression statements can be translated into C++ code without any change.

Methods in the MLM class can be aggregation operators defined by the Arden Syntax, such as *Min & Increase,* that are applied to the data. Since the number of parameters in the list for these aggregation operators may vary, our approach for parameter passing is to assign the list to an object of the MLM class by means of operators, which are , & =. In this case the parameters passing for the operators are fulfilled through the object.

According to the Arden Syntax, the action slot is executed immediately if the expression concludes true. By means of a *while* loop with break statements, it ensures that the conclude statement ends execution in the logic slot. An error flag set by class methods is used for checking whether enough data is available or unexpected data types are found when the action slot is executed.

To implement the pre-compiler according to the translation schema, two text files are used during the translation phase. These are a 'declaration table' for variable definition and a 'knowledge table' for the data, logic and action slot. After the MLM pre-compiler has gone through the MLM, the files are tailored into a file with the same name as the MLM itself but with a different extension name. In our case the extension name is *cxx*.

3.2.2. Evoke schema

An MLM can be triggered by any of the following statements:

(i) occurrence of some events;
(ii) a time delay after an event;
(iii) periodically after an event;
(iv) a direct call from another MLM or a program.

There are different types of event, such as an update or insertion in a patient data-base, a medically relevant occurrence or an institution-specific occurrence. An event statement in the data slot assigns a variable to a specific event. The variable can be used in the evoke slot. The time of occurrence of an event is the value of the event variable. The evoke slot defines how an MLM can be triggered according to evoke types 1–3 above. These are the simple trigger statement, delayed trigger statement, and periodic trigger statement. The MLM pre-compiler translates the evoke parameters in the data slot and the evoke statements in the evoke slot according to the following strategies:

— Creating an information table that contains

two keywords on each line. The first word identifies the type of trigger and the other is a parameter name in the patient data-base. This table is used by the data-base management system to set up triggers in the patient data-base [13] (see Fig. 3).

— Producing another definition table to be used for scheduling of the MLMs at run time. The table includes variable name, MLM name, start time, duration, every interval time and an evoke flag (see Fig. 3).

The evoke slot does not specify the fourth kind of triggering, calling of an MLM by another MLM. Triggering is defined in the MLM statement within the data slot. The MLM's name can be derived from the statement or from the event variable. There are two different types of MLM call. One is when one MLM calls another MLM without any optional delay, which means that the call is synchronous. In this case the names of called MLMs are put into the current calling MLM during the pre-compilation process. At run time the named MLMs can be called directly. In the other case the call is asynchronous, as the named MLM is called with a delay when the calling MLM terminates, for example *Call mlm_1 delay 3 days.* The strategy for handling asynchronous calls is as follows. The information about the called MLM names and the specified duration found during pre-compilation are stored in an evoke definition table. At run time the called MLM names with the delay period are loaded into an execution queue.

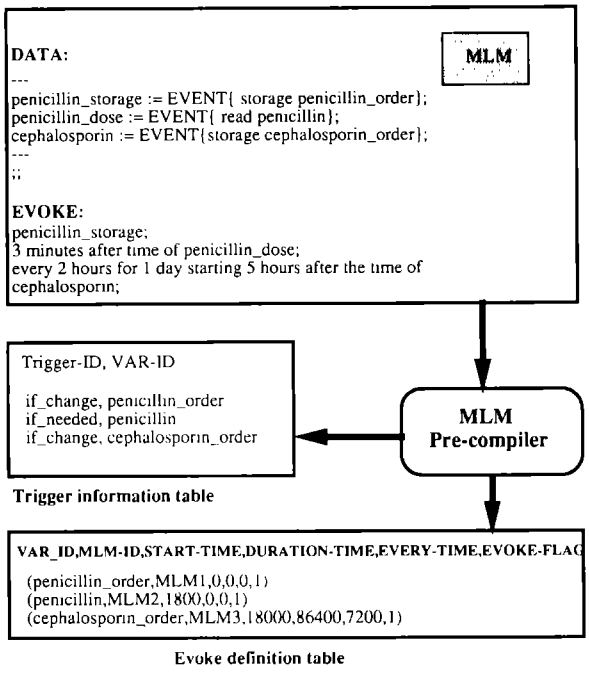

Fig. 3. Examples of evoke definition and trigger information tables generated during pre-compilation. Trigger mechanisms available within the DBMS may vary, but the basic trigger type needed is if_change or if_update, allowing the data-driven mechanism to be implemented. The evoke definition table is used by the MLM engine when scheduling execution of different MLMs. START-TIME, DURATION-TIME and EVERY-TIME in the evoke definition table are expressed in seconds.

4. Implementation of the MLM pre-compiler

The MLM pre-compiler operates in phases, each of which transforms the source program from one representation to another. The compilation process performs decomposition of the tokens, MLM parsing, highlighting syntax errors and creation of C++ code. A decomposition of the MLM pre-compiler is shown in Fig. 5.

As the principles and techniques of compiler construction are well documented, writing code for a compiler is a simple but time-consuming job. It is, therefore, beneficial to apply a compiler tool that can reduce the burden of programming [14]. The compiler construction tools support the automatic generation of compilers for imperative programming language. With respect to efficiency the tools are competitive with programming by hand. The toolbox from Karlsruhe University was used to construct our MLM compiler [15]. The toolbox has more functionality and requires less coding than the well-known tools 'Lex' and 'Yacc'.

4.1. A toolbox for compiler construction

Karlsruhe University has constructed a set of tools for all phases of high-speed compiler generation. A series of papers on these tools have been

published in different computer journals and conferences [15].

Tools that have been used for the MLM compiler are generators for lexical analysers (Rex), syntax analysers (Lalr), and abstract syntax trees and attribute evaluators (Cg). The structure of the MLM pre-compiler and the tools used within the toolbox are shown in Fig. 4. The formalism of MLM.pars is used to describe the parser, which is an extension of the input language for abstract syntax tree and attribute evaluator tools. The MLM.rex, MLM.lalr and MLM.cg express four phases of the MLM pre-compiler: lexical analyzer, syntax analyzer, semantic analyzer, and intermediate code generator. Finally, the C code generated from the specifications, the reusable modules from the toolbox and C functions specified by users are compiled and linked with a RISC C compiler. The result is the MLM pre-compiler. The work has been done on a DEC 5000/200 with the Ultrix operating system.

4.2. *The pre-compilation process*

The first phase of the MLM pre-compiler does a complete lexical analysis of the Arden Syntax

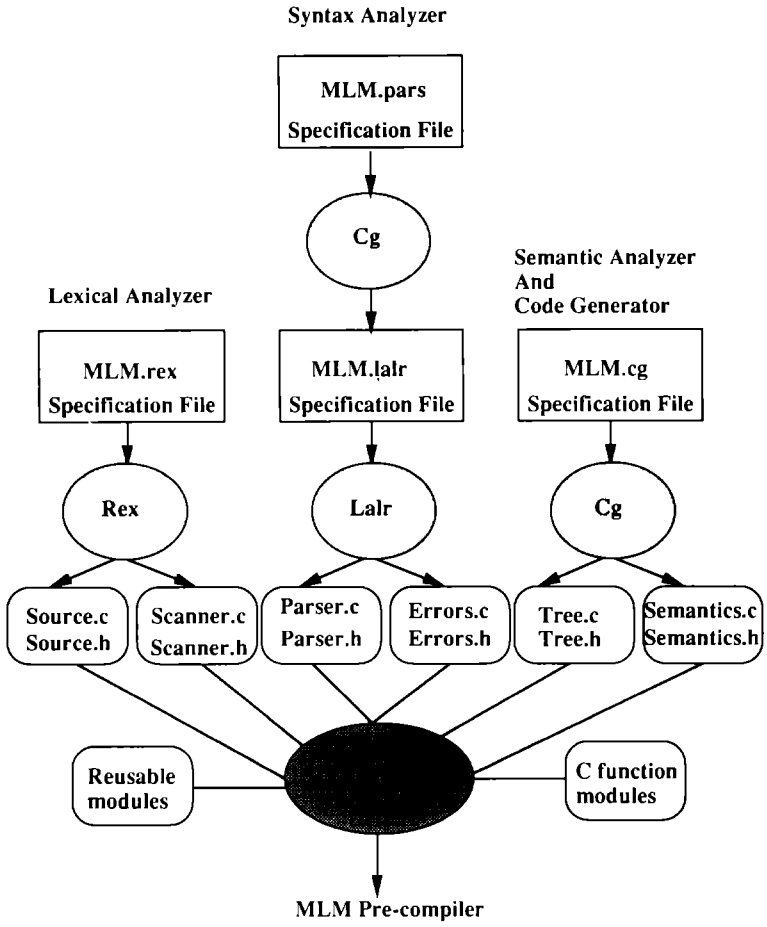

Fig. 4. Construction of the MLM pre-compiler using the Karlsruhe toolbox.

specification. The syntax analysis program involves grouping the tokens of the MLM source code into grammatical phases that are used by the compiler to synthesise output. A parse tree that describes the syntactic structure of the input is produced from it. The context-dependent errors are handled, e.g. unbalanced parentheses and missing statement terminators. After the semantic analysis phase, the compiler transforms the specification into an internal form called abstract syntax tree. The tree is a compressed representation of the parse tree in which the operators appear as the interior nodes and the operands of an operator are the children of the node for that operator. The tree corresponds to the syntactic structure of the statement parts in the program and additional nodes. These nodes represent declared objects on which the statements operate. The fourth phase of the compiler performs structural transformations on the generated syntax tree and does some high-level optimisation. Finally, the C++ code, evoke definition and trigger table are produced. A symbol table is used to store objects of the MLM class, constants, etc.

5. Run time environment

During the pre-compilation process, C++ functions, evoke definition and trigger information tables are generated. The C++ functions are compiled into object files, which are stored in a library called MLMLIB. The DSS is finally generated by linking MLMLIB together with the MLM engine and a library of Arden Syntax operators (LASO). LASO, which is implemented as methods in the *mlm_type* class, contains all data operators defined in the syntax, such as mathematical functions and query aggregations. The inference mechanism is controlled by the MLM engine, which, during run time, receives trigger messages and executes

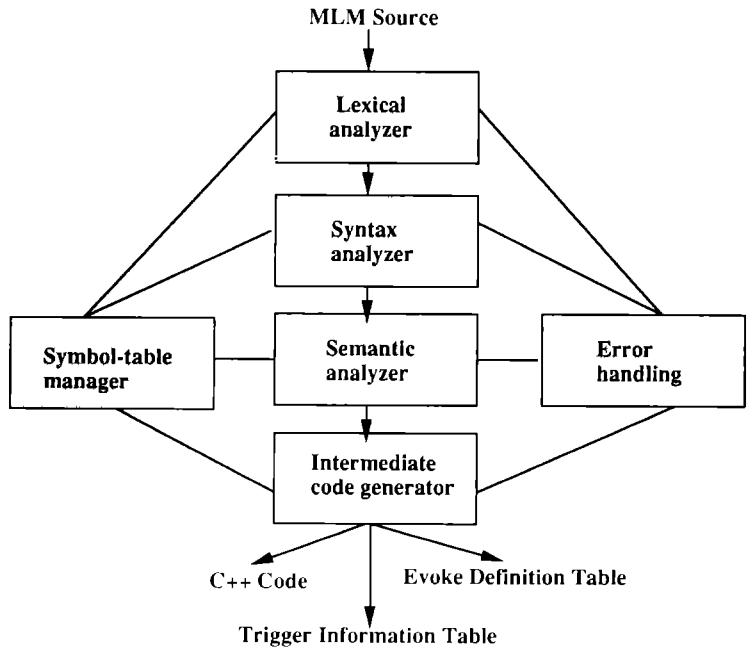

Fig. 5. Phases of the MLM pre-compiler.

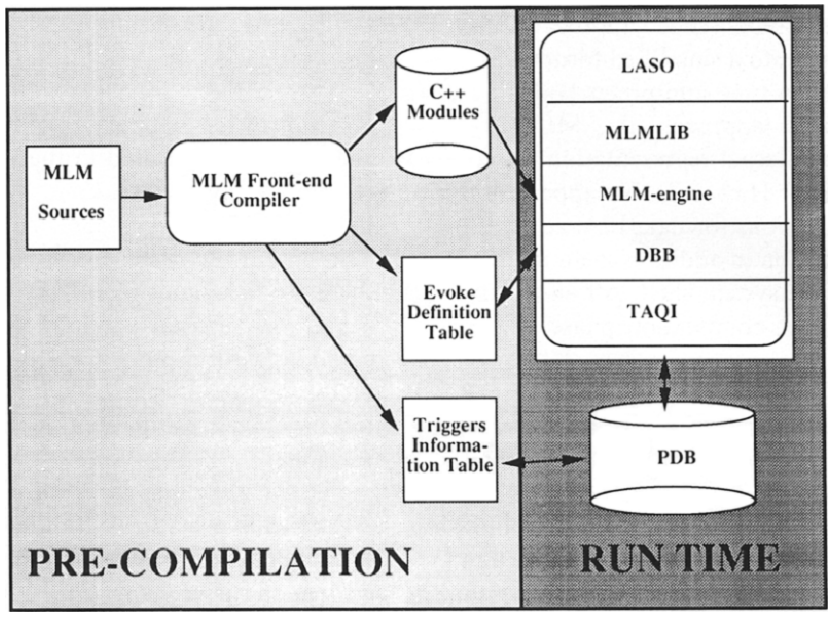

Fig. 6. MLM-based DSS working environment.

MLMs according to their time specifications in the evoke table.

A prototype critiquing system for anticoagulant therapy has been developed with the described method [6]. Anticoagulant therapy as a short term of prophylaxis for patients with acute venous thrombosis or pulmonary embolism is critical in that the therapeutic window of anticoagulant drugs is very narrow and an inaccurate dose might have a life-threatening effect. The application area is fit for computerised decision support. The prototype DSS is evoked through a specialised data entry system taking the role of a patient data-base. Fig. 6 shows the relation between the pre-compilation phase and the run time environment together with a decomposition of the DSS into its different parts.

In a complete system a data-base bridge (DBB) between the DSS and heterogeneous patient databases is required as well as a trigger, action and query interface (TAQI). The DBB is required for mapping of data-base queries expressed in the MLM to the query language of the specific database connected to the DSS. The TAQI is needed for communication and message passing between the DSS and the patient data-base (PDB).

6. Discussion

Much research activity has been focused on tools for building domain-specific knowledge bases. Work to specify a standard knowledge representation format for sharing of medical logic modules may significantly reduce the burden of constructing knowledge bases in different medical domains, but several challenges still remain. One important problem addressed in this paper involves the implementation issues of such a standard. The approaches for translation, implementation and execution of MLMs using C++ have been presented. Important problems and solutions concerning how to deal with data types, query aggregation operators, MLM calls and evoke criteria have been discussed. The use of the Karlsruhe compiler toolbox has resulted in a fast and easy-to-maintain pre-compiler.

Some alternatives for the realisation of a DSS based on MLMs other than the C++ approach are

possible. In the pseudo-code approach, each module is compiled into a simplified form that can be executed by a run time interpreter [16]. In the expert system shell approach the MLMs are translated into an internal representation of some expert system shells. However, this approach requires external functions for data-base access and trigger communication in addition to an inference engine of the expert system shell. All alternatives have two phases: pre-compilation phase and run time phase.

The first is the pre-compilation phase, where the MLM syntax is checked by a parser and an intermediate form is generated, and all information referring to triggers and evoke criteria is collected and stored separately. The nature of the object-oriented language, such as the class construct and data encapsulation, constructor and destructor, private and public sections, objects and messages, friends, overloading of operators and methods, makes the pre-compiled code similar to the MLM source, so that the translation work in the C++ approach is made easier compared with the other approaches. In the expert system case, the first pre-compilation phase may be more or less problematic depending on the expressiveness of the internal representation format.

The second phase is the run time phase. As speed performance mostly depends on the data-base access, minor differences in execution time for the MLM logic are irrelevant. All approaches are also required to solve access and query mapping problems to at least one data-base system. For the pseudo-code approach an interpreter that can execute pseudo-codes must be written. However, for other approaches some commercial tools can be employed, such as the C++ compiler, the debugger and the inbuilt inference engine in the expert system shell. Moreover, an MLM engine that communicates with patient data-bases and executes evoked MLMs is necessary for the C++ approach.

There are several reasons for choosing C++ in general:

— Support for object-oriented programming is the most significant property of C++.
— The use of C++ simplifies the compilation work, and maintenance of the pre-compiler becomes easier.
— Data-base connection is straightforward, especially when an object-oriented data-base management system is used in the DSS environment.
— Portability.

In order to reduce memory requirements in case there are thousands of MLMs in a large decision support system, a dynamic library linking method could be used in our approach. A program based on this method dynamically links those MLM object files that are requested by the MLM engine at run time.

The work presented is part of the European AIM/Helios project, where the described method for building DSS will be part of a medical software engineering environment [17]. Within the Helios project, the object-oriented data-base GemStone is used together with Decstations running the Ultrix operating system. Work with the described DSS building method is also continuing on PC/dBase and Informix platforms. Further work is also needed in relation to DBB and TAQI as well as with tools for knowledge management, adaptation, simulation and debugging of MLMs.

7. Acknowledgement

We would like to thank Dr. Sunil Kohli for his useful comments concerning the paper. This work is part of the research within the EEC AIM project Helios-2 (A2015) carried out jointly by Broussais University Hospital, the German Cancer Research Center, Geneva University State Hospital, Cap Sesa Tertiaire, CMD (Uppsala University), Linköping University, and Digital Equipment Corporation, and is partly supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

8. Appendix A. MLM class description

```
/*******************************************************/
class mlm_type
{
private:
  TYPE_MLM *mlm[MAX_ELEMENTS_IN_MLM]; // array of values
and time
  int nr_values;
  int nr_alloc;
```

```
public:

  mlm_type();
  mlm_type(int);
  ~mlm_type();        // destructor

  /* Get data (for data-slot) */
  int read(int,char *,char *);
  int read_last(int,char *,char *);
  void read_exist(int,char *,char *);

  void print(char *);

  char * value(char *);
  NUMERICAL_TYPE value(NUMERICAL_TYPE);
  BOOLEAN_TYPE value(BOOLEAN_TYPE);

  BOOLEAN_TYPE is_null(void);
  BOOLEAN_TYPE is_notnull(void);

  // operator overloading
  mlm_type& operator =(mlm_type&);
  mlm_type& operator =(BOOLEAN_TYPE);
  mlm_type& operator =(NUMERICAL_TYPE);
  mlm_type& operator =(char *);
  mlm_type& operator , (BOOLEAN_TYPE);
  mlm_type& operator , (NUMERICAL_TYPE);
  mlm_type& operator , (char *);
  mlm_type& operator , (mlm_type&);
  friend mlm_type &operator &&(mlm_type&,mlm_type&);
  friend mlm_type &operator ||(mlm_type&,mlm_type&);
  friend NUMERICAL_TYPE operator +
  (mlm_type&,NUMERICAL_TYPE);
  friend NUMERICAL_TYPE operator +
  (NUMERICAL_TYPE,mlm_type&);
  friend mlm_type &operator + (mlm_type&,mlm_type&);
  friend NUMERICAL_TYPE operator -
  (mlm_type&,NUMERICAL_TYPE);
  friend NUMERICAL_TYPE operator -
  (NUMERICAL_TYPE,mlm_type&);
  friend mlm_type & operator -
  (mlm_type&,mlm_type&);
  friend NUMERICAL_TYPE operator /
  (mlm_type&,NUMERICAL_TYPE);
  friend NUMERICAL_TYPE operator /
  (NUMERICAL_TYPE,mlm_type&);
  friend NUMERICAL_TYPE operator /
  (mlm_type&,mlm_type&);
  friend NUMERICAL_TYPE operator *
  (NUMERICAL_TYPE,mlm_type&);
  friend mlm_type & operator * (NUMERICAL_TYPE,mlm_type&);
  friend NUMERICAL_TYPE operator * (mlm_type&,mlm_type&);
  friend int operator != (mlm_type&,char *);
  friend int operator != (mlm_type&,BOOLEAN_TYPE);
  friend int operator != (mlm_type&,NUMERICAL_TYPE);
  friend int operator != (mlm_type&,mlm_type&);
  friend int operator == (mlm_type&,char *);
  friend int operator == (mlm_type&,NUMERICAL_TYPE);
  friend int operator == (mlm_type&,BOOLEAN_TYPE);
  friend int operator == (mlm_type&,mlm_type&);
  friend int operator < (mlm_type&,NUMERICAL_TYPE);
  friend int operator < (mlm_type&,mlm_type&);
  friend int operator > (mlm_type&,NUMERICAL_TYPE);
  friend int operator > (mlm_type&,mlm_type&);
  friend int operator <= (mlm_type&,NUMERICAL_TYPE);
  friend int operator <= (mlm_type&,mlm_type&);
  friend int operator >= (mlm_type&,NUMERICAL_TYPE);
  friend int operator >= (mlm_type&,mlm_type&);
  friend mlm_type& operator | (mlm_type&,mlm_type&);
  friend mlm_type& operator | (char *,mlm_type&);
  friend mlm_type& operator | (mlm_type&,char *);
  friend int operator << (mlm_type&,mlm_type&);
  friend mlm_type& decrease (mlm_type&);
  friend mlm_type& avg(mlm_type&);
  friend mlm_type& min(mlm_type&);
  friend mlm_type& max(mlm_type&);
  friend mlm_type& last(mlm_type&);
  friend mlm_type& first(mlm_type&);
  friend mlm_type& last_num(mlm_type&,mlm_type&);
  friend mlm_type& first_num(mlm_type&,mlm_type&);
  friend mlm_type& max_num(mlm_type&,mlm_type&);
  friend mlm_type& min_num(mlm_type&,mlm_type&);
  friend mlm_type& count(mlm_type&);
  friend mlm_type& abs(mlm_type&);
  friend mlm_type& exist(mlm_type&);
  friend BOOLEAN_TYPE any_call(mlm_type&);
  friend BOOLEAN_TYPE all (mlm_type& );
  friend BOOLEAN_TYPE  no (mlm_type& );
  friend mlm_type& variance(mlm_type&);
  friend mlm_type& stddev(mlm_type&);
  friend mlm_type& median(mlm_type&);
  friend mlm_type& increase(mlm_type&);
  friend mlm_type& percent_increase(mlm_type&);
  friend mlm_type& percent_decrease(mlm_type&);
  friend mlm_type& nearast(mlm_type&,mlm_type&);
  friend double exp_mlm(mlm_type&);
  friend double sqrt_mlm(mlm_type&);
  friend double sin_mlm(mlm_type&);
  friend double arcsin_mlm(mlm_type&);
  friend double cos_mlm(mlm_type&);
  friend double arccos_mlm(mlm_type&);
  friend double tan_mlm(mlm_type&);
  friend double arctan_mlm(mlm_type&);
  friend double log_mlm(mlm_type&);
  friend double log10_mlm(mlm_type&);
  friend mlm_type& int_mlm(mlm_type&);
  friend mlm_type& exist(mlm_type&);
  friend mlm_type& sum(mlm_type&);

  friend void write(mlm_type&);
  friend int get_data(mlm_type&,int,char *,char *);

  // to handle mlm return values
  friend mlm_type& call_mlm(char *,char *);
  friend void return_mlm(mlm_type&);
  friend mlm_type& year(mlm_type&);
  friend mlm_type& month(mlm_type&);
  friend mlm_type& week(mlm_type&);
  friend mlm_type& day(mlm_type&);
  friend mlm_type& hour(mlm_type&);
  friend mlm_type& minute(mlm_type&);
  friend mlm_type& second(mlm_type&);
  friend mlm_type& now(void);

  friend mlm_type& time_call(mlm_type&);
  friend BOOLEAN_TYPE Timecomp(mlm_type&,short,mlm_type&);
  friend mlm_type& within_sur(mlm_type&,mlm_type&);
  friend mlm_type& within_pre(mlm_type&,mlm_type&);
  friend mlm_type& within_fol(mlm_type&,mlm_type&);
  friend mlm_type& within_to(mlm_type&,mlm_type&);
  friend mlm_type& ago(mlm_type&);
  friend mlm_type& after_call(mlm_type&,mlm_type&);
  friend mlm_type& before_call(mlm_type&,mlm_type&);
  friend mlm_type& iso_time(char *);
  friend mlm_type& where(mlm_type&,mlm_type&,short,mlm_type&);
  friend mlm_type& give_to_temp(mlm_type&,int,int);

  BOOLEAN_TYPE bool(void);
  BOOLEAN_TYPE present(void);
};

/*****************************************************/

struct mlm_type_struct
{
  char type;      //N,S,B,T,D,d
```

```
  union
  {
     NUMERICAL_TYPE nvalue;
     STRING_TYPE svalue;
     BOOLEAN_TYPE bvalue;
     TIME_TYPE tvalue;
     DURATION_TYPE dvalue;
   };
   TIME_TYPE vtime;      // primary time of data
};

typedef struct mlm_type_struct TYPE_MLM;

/*********************************************************/
```

9. Appendix B. The C++ code for the operator Min

```
mlm_type& min(mlm_type& a)
{
int s=1,j=1,k=1,t=1,index;
NUMERICAL_TYPE tmp=0.0;
mlm_type *temp_obj=get_temp_object();
if (a.nr_values==0){
      temp_obj->nr_values=0;
      error_in_mlm("Warning: No Elements in min");
}
else {
      for (int i=0;i<a.nr_values-1;i++)
      {
            if(a.mlm[i]->type=='N')
            j++;
            if(a.mlm[i]->type=='d'||a.mlm[i]->type=='D')
            k++;
            if(a.mlm[i]->type=='T')
            t++;
            if(a.mlm[i]->type=='S')
            s++;
      }
     if(j==a.nr_values){
           tmp=a.mlm[0]->nvalue;
           for (int i=0;i<a.nr_values-1;i++){
           if(a.mlm[i]->nvalue<tmp){
             tmp=a.mlm[i]->nvalue;
             index=i;
           }
      }
      temp_obj->mlm[0]->nvalue=tmp;
      temp_obj->nr_values=1;
      temp_obj->mlm[0]->type='N';
      temp_obj->mlm[0]->vtime=a.mlm[index]->vtime;
}
else if(k==a.nr_values){
      tmp=a.mlm[0]->dvalue;
      for (int i=0;i<a.nr_values-1;i++){
            if(a.mlm[i]->type=='D')
              a.mlm[i]->dvalue=a.mlm[i]->dvalue*26297
            if(a.mlm[i]->dvalue<tmp){
              tmp=a.mlm[i]->dvalue;
              index=i;
            }
      }
      temp_obj->mlm[0]->dvalue=tmp;
           temp_obj->mlm[0]->type='d';
           temp_obj->nr_values=1;
           temp_obj->mlm[0]->vtime=a.mlm[index]->vtime;
     }
     else if(t==a.nr_values){
           tmp=a.mlm[0]->tvalue;
           for (int i=0;i<a.nr_values-1;i++){
                if(a.mlm[i]->tvalue<tmp){
                  tmp=a.mlm[i]->tvalue;
                  index=i;
                }
           }
           temp_obj->mlm[0]->tvalue=tmp;
           temp_obj->mlm[0]->type='T';
           temp_obj->nr_values=1;
           temp_obj->mlm[0]->vtime=a.mlm[index]->vtime;
     }
     else {
          temp_obj->nr_values=0;
          error_in_mlm("Semantic error: in min");
     }
}
return *temp_obj;
}
```

10. References

[1] P. Clayton, T. Pryor, O. Wigertz and G. Hripcsak, Issues and structures for sharing medical knowledge among decision-making systems, in Proc. 13th SCAMC, IEEE Computer Society Press, eds. C. Lawrence, I.I.I. Kingsland, pp. 116–121 (IEEE Computer Society Press, Washington, 1989).

[2] G. Hripcsak, P. Clayton, T. Pryor, P. Haug, O. Wigertz and J. van der Lei, The Arden Syntax for medical logic modules, in Proc 14th SCAMC, IEEE Computer Society Press, eds. Randolph A. Miller, pp. 200–204 (IEEE Computer Society Press, Los Alamitos, CA, 1990).

[3] M. Musen, Dimensions of knowledge sharing and reuse, Comput. Biomed. Res. 25 (1992) 435–467.

[4] O. Wigertz, K. Arkad, S. Chowdhury, G. Magyar, X. Gao and H. Åhlfeldt, Standards for medical knowledge representation, in Proc. IMIA Working Conference on Hospital Information Systems, pp. 155–162 (North Holland, IMIA Series, 1991).

[5] American Society for Testing and Materials, Standard Specification for Defining and Sharing Modular Health Knowledge Bases: Arden Syntax for Medical Logic Modules (Standard E1460, October 1991).

[6] HELP Frame Manual, Version 1.6 (LDS Hospital, Salt Lake City, 1989).

[7] C. McDonald, L. Blevins, W. Tierney and D. Martin, The Regenstrief medical records. MD Comput. 5 (5) (1988) 34–47.

[8] O. Wigertz, N. Shahsavar, H. Gill, X. Gao, K. Jönsson, K. Arkad, P. Ohlsson and H. Åhlfeldt, Knowledge representation for an anticoagulant therapy advisor, in Proc. 11th MIE, pp. 74–79, 1993.

[9] X. Gao, N. Shahsavar, K. Arkad, H. Åhlfeldt, G. Hripcsak and O. Wigertz, Design and functions of medical knowledge editors for the Arden Syntax, in 7th World Congress on Medical Informatics, eds. K.C. Lun, P. Degoulet, T.E. Piemme and O. Rienhoff, pp. 472–477 (Elsevier Science, Amsterdam, 1992).

[10] K. Arkad, G. Hans, U. Ludwig, N. Shahsavar, X. Gao and O.Wigertz, Medical logic module (MLM) representation of knowledge in a ventilator treatment advisory

system, Internat. J. Clin. Monitor. Comput. 8 (1991) 43–48.

[11] B. Johansson and O. Wigertz, An object-oriented approach to interpret medical knowledge based on the Arden Syntax, in Proc. 16th SCAMC, eds. Mark E. Frisse, pp. 52–56 (MacGraw Hill, New York, 1992).

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

[13] S. Chakravarthy, S.B. Navathe, S. Garg, D. Mishra and A. Sharma, An evaluation of active DBMS development, in UF-CIS Technical Report TR-90-23 (Database Systems Research and Development Center, Computer and Information Sciences Department, University of Florida, Gainesville, 1990).

[14] A. Aho, R. Sethi and J. Ullman, Compilers: Principles, Techniques, and Tools. (Addison-Wesley, Reading, MA, 1986).

[15] J. Grosch and H. Emmelmann, A tool box for compiler construction, in Compiler Generation Report No. 20 (GMD Forschungsstelle an der Universität Karlsruhe, January 1990).

[16] G. Hripcsak, C. James, J. Stephen and P.D. Clayton, The Columbia Presbyterian Medical Center decision-support system as a model for implementing the Arden Syntax, in Proc. 15th SCAMC, IEEE Computer Society Press, eds. Paul D. Clayton, pp. 248–252 (MacGraw Hill, New York, 1991).

[17] M. Jaulent, D. Sauquet and P. Degoulet, The interface manager of the HELIOS medical software engineering environment, in 7th World Congress on Medical Informatics, eds. K.C. Lun, P. Degoulet, T.E. Piemme and O. Rienhoff, pp. 1305–1310 (Elsevier Science, Amsterdam, 1992).