Pergamon

Comput. Biol. Med. Vol. 24, No. 5, pp. 365–370, 1994
Copyright © 1994 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0010–4825/94 $7.00 + 0.00

0010-4825(94)00023-9

A C++ FRAMEWORK FOR DEVELOPING MEDICAL LOGIC MODULES AND AN ARDEN SYNTAX COMPILER

RONALD A. KUHN AND ROGER S. REIDER
IBM Corporation, 6700 Rockledge Drive, Mail Drop 45A4, Bethesda, MD 20817, U.S.A.

(*Received* 3 *June* 1994; *received for publication* 21 *July* 1994)

Abstract—When developing a clinical decision support system that uses knowledge expressed in Arden Syntax, the availability of a robust means of translating Arden Syntax into an executable module becomes critical. This paper describes an approach where Arden Syntax is translated into an intermediate pseudo-Arden language that is in turn compiled and linked to create the executable module. The pseudo-Arden language is defined in C++ using specialized class libraries and preprocessor macros. This approach provides an alternative means of developing the code generator for an Arden Syntax compiler.

C++ Framework Arden Syntax Compiler Object-oriented
Medical logic module (MLM) Medical logic

INTRODUCTION

One of the first decisions confronting organizations embarking on the development of a clinical decision support system is "how shall the medical knowledge be represented in an operational system?" There are numerous choices: Prolog, Lisp, C/C++, PL/1, etc. While these are good languages they lack many of the facilities of specific value in the clinical medical domain. In particular, a well-developed concept of time as a fundamental attribute of data is invaluable.

Arden [1] is a newcomer, but has many attractive properties: a well-developed concept of primary time being one of them, the potential for portability, and reuse are others. Its acceptance in the medical community (in terms of the number of operational users of Arden) is small at present, but growing. In the longer term, it is conceivable that a large body of clinical rules written in Arden may become generally and publicly available once Arden Syntax authoring facilities become more widely available.

Assuming that Arden is the language of choice, the next question to be answered is "how are rules written in Arden Syntax translated into the executable instructions of the underlying computer system?" If a compiler approach is chosen, then a number of design issues must be addressed. Fortunately, compiler construction tools (e.g. Lexx/Yacc) can be used to develop the lexical analysis and parsing functions with minimal difficulty.

The complexity of the code generation function, however, can vary significantly depending upon design approach used. Translating Arden-written rules directly into machine code probably has the best potential for creating high performance executable code. Generating "pCodes" is another alternative that is successfully operating at Columbia-Presbyterian Medical Center [2].

Another approach, pursued by the authors, is to translate into an intermediate high-level language which is then compiled to produce the executable Medical Logic Module (MLM). The design of the compiler's code generator is simplified by allowing it to work with higher level abstractions and simpler instruction sequences. With this approach, the difficult and complex issues of generating optimized machine instructions, linkers, debuggers, and runtime support libraries is addressed by the underlying compiler product.

OVERVIEW OF C++ APPROACH

The approach investigated by the authors translates MLMs written in Arden Syntax into a high-level pseudo-Arden language expressed in C++ through a series of specialized class libraries and preprocessor macros. Our primary objectives in designing the C++ classes were to simplify the development of the code generation portion of our Arden compiler by (1) enabling a very literal translation between Arden Syntax and the resulting C++ code; and (2) providing a framework for abstracting and encapsulating the intricacies of Arden MLMs and their *logic slot* implementation.

Several major components are required to address the basic structure of medical logic modules (MLMs) and operational use of Arden variables:

- Class libraries to extend the basic C++ language to more directly address the expression of Arden Syntax and the manipulation of Arden variables.
- Preprocessor macros to simplify further the expression of Arden Syntax.
- Class libraries to define the runtime interfaces to MLMs.
- A set of run-time services to provide rule logic with access to facilities outside of the language itself such as: access to data and action slot processing.

Arden variables represent a significant amount of function in the Arden Syntax. Their design and capabilities are critical to a successful implementation of Arden MLMs. A basic design approach to their definition in C++ has been demonstrated by Johansson and Wigertz at Linkoping University, Sweden [3]. Our implementation addresses all of the features outlined in the current Arden standard and enables the use of Arden variables as regular C++ variables. Specifically:

- All 94 operations defined on Arden variables are supported.
- Support for dynamic runtime typing through polymorphic behaviors.
- Support for primary time.
- Reference counting for improved memory management.
- Support for unbounded list variables.

The translated MLMs themselves are abstracted into an object. The design provides methods for: identification, evocation, data acquisition, logic, and action slot processing. The compiler uses the same empty class skeleton for all MLMs. Since each MLM is compiled and linked into a separated shared library, duplicate symbol problems are avoided. The MLM framework class provides applications a consistent mechanism for properly executing the MLM.

ARDEN VARIABLE IMPLEMENTATION

Our basic approach is to fully encapsulate all Arden variable behaviors in an enveloping class. This enables program logic to work with all variables in a consistent manner without regard to data type or how they are allocated (i.e. dynamic, automatic, or static). In C++, user-defined data types are referred to as classes. Classes consist of an aggregate of data elements and a set of operations (member functions) designed to manipulate the data. Through inheritance, specialized classes can be defined by implementing behaviors that augment or replace those in the super class. Arden variables, the data in the MLMs, are defined by the class structure illustrated in Fig. 1.

ARDEN SYNTAX OPERATORS

The Arden Variable classes, shown in Fig. 1, contain 94 member functions that implement all of the operations defined for Arden variables in the ASTM standard (1). To make the translation between Arden Syntax and C++ as easy as possible, the names of the member functions resemble their corresponding Arden operators. Where appropriate, C++ operators like +, < and == were defined to implement their Arden counterparts. Table 1 lists some of the Arden Syntax operators along with their corresponding C++ member functions.

During the execution of the MLM knowledge slots, two Arden variables are always accessible: **now** (a time constant indicating the time of execution) and **eventTime** (a time

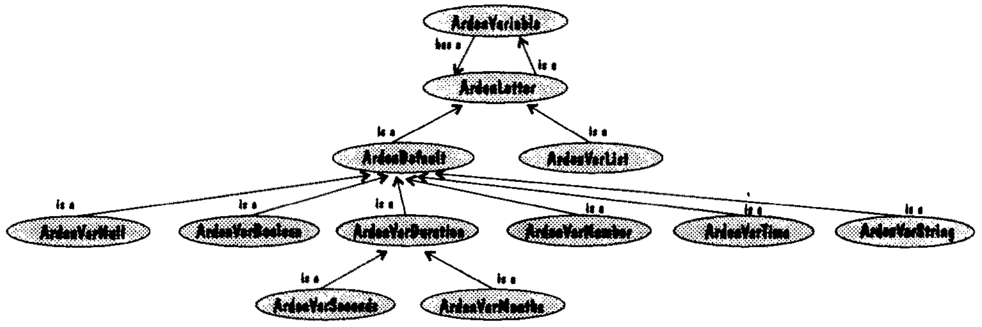

Fig. 1. C++ class relationship and hierarchy for Arden Syntax variables.

constant indicating the time that the evoking event occurred). Three Arden operators require access with the **now** variable: IS WITHIN PAST, OCCUR WITHIN PAST and AGO. These three operators are implemented in C++ using other Arden operators with the **now** variable as one of the parameters: IS WITHIN <duration> PRECEDING **now**, occur within <duration> PRECEDING **now** and <duration> BEFORE **now**, respectively.

DYNAMIC RUNTIME TYPING

The type of an ArdenVariable is not defined in the MLM, but is established when the variable is first used. The support for dynamic typing at runtime for ArdenVariables is implemented in C++ by using a variation of the envelope/letter design idiom [4]. In our implementation, the envelope class is the ArdenVariable and the letter class (which is also derived from the ArdenVariable) is an ArdenLetter. Figure 2 contains a partial listing of the ArdenVariable class definition with some of the Arden operators defined.

The envelope class fully encapsulates the letter class. Access to the letter object occurs only through the envelope class. As shown in Fig. 2, the envelope class contains a private data element that is the address of a letter object. To implement dynamic run-time, one of the specialized letter subclasses (e.g. ArdenVarBoolean, ArdenVarSeconds) is dynamically allocated depending on the specific type of ArdenVariable required.

Each Arden operator defined in the ArdenVariable class forwards its call to its letter object as shown by the ArdenVariable::isAfter method in Fig. 3. The logic of the Arden operation is performed by the letter object as shown by the other methods illustrated in Fig. 3. The ArdenLetter class is an abstract base class that specifies the interface of the ArdenLetter subclasses. Each subclass implements its logic by overriding corresponding virtual functions in the superclass. This polymorphic behavior enables the program to manipulate many different types of letter objects without regard to the details of the specific ArdenLetter subclass actually in use.

Table 1. Comparison of selected arden Syntax and their C++ translations

Arden Syntax operator	C++ translation
aList := aVar1, aVar2	aList = (aVar1, aVar2)
aBool := aBool1 OR aBool2	aBool = aBool1 OR aBool2
aBool := aVar1 = aVar2	aBool = aVar1 == aVar2
aBool := aTime1 IS AFTER aTime2	aBool = aTime1.isAfter (aTime2)
aResult := aVar1 + aVar2	aResult = aVar1 + aVar2
aTime := aDur AGO	aTime = aDur.before (now)
aNum := STDDEV aList	aNum = aList.stddev ()
aList := MINIMUM aNum FROM aList1	aList = aNum.minimumFrom (aList1)
aTime := TIME aVar	aTime = aVar.time ()

```
class ArdenVariable
  {
   public:
     // constructors
     ArdenVariable(ArdenLetter*);        // to add an envelope to a letter
     ArdenVariable(const ArdenVariable&);// copy constructor
     ...
     // Arden Syntax operators
     virtual ArdenVariable operator,(const ArdenVariable&) const;
     ...
     virtual ArdenVariable stddev() const;
     ...
     virtual ArdenVariable time() const;

   protected:
     ArdenVariable();       // default constructor for letter construction

   private:
     ArdenLetter *letter;   // address to corresponding letter
     short refCount;        // reference count of letter
  };
```

Fig. 2. ArdenVariable (envelope) class definition.

REFERENCE COUNTING OF LETTER OBJECTS

A significant amount of time in C++ programs can be spent managing dynamic (a.k.a. heap) memory. To improve the performance of using dynamically allocated ArdenLetter objects, the ArdenVariable class implements reference counting to share them between multiple envelopes. This also helps eliminate potential memory leaks that may result when using dynamically allocated objects carelessly.

As shown in Fig. 2, the ArdenVariable class contains a private data element, refCount, that is used by the letter to track the number of envelopes that contain the address of this letter. When an ArdenLetter object is dynamically allocated, the default constructor, ArdenVariable(), is called to initialize the refCount to 0 (no envelopes). The letter object is inserted into an envelope using the ArdenVariable(ArdenLetter*) constructor. This function saves the address of the letter object and increments the letter's refCount. When the envelope object is deleted, the contained letter's refCount is decremented and checked to see if any other envelopes contain this letter. If no such envelopes exists (refCount = = 0), the letter object is also deleted.

There are two other methods that utilize refCount: the copy constructor and assignment operator. The copy constructor, ArdenVariable(const ArdenVariable&), is used when an ArdenVariable is created from another ArdenVariable. The assignment operator, operator = (const ArdenVariable&), is used when the value of a variable is copied into an existing variable. In both cases, the address of the letter object is copied and the letter's refCount is incremented. No new letter objects are created. In the case of the assignment operator, the target variable's (lvalue) letter object may be deleted if it is not used in any other envelopes (ArdenVariables).

CODE REUSE USING C++ INHERITANCE

As shown in Fig. 2, the ArdenDefault class is super class for all single element Arden variables. The ArdenDefault class contains methods for all Arden operators not implemented by the subclasses. Arden operations not implemented in a subclass are handled by methods provided in the ArdenDefault base class.

Figure 3 provides the complete C++ implementation of the Arden **IS AFTER** operator for all the Arden variable classes. According to the Arden Syntax standard, the **IS AFTER** operator is only valid between Arden time variables. As illustrated by the figure, the **isAfter** method is not implemented in all the classes. The majority of the classes resolve the **IS AFTER** operation by calling the **ArdenDefault::isAfter** method which returns a null Arden variable. The ArdenVarTime class implements the **IS AFTER** operation by calling the greater than (>) operator.

Implicit with every single-element Arden variable is an associated primary time. The ArdenDefault class implements the methods and data necessary to support primary time

```
ArdenVariable ArdenVariable::isAfter(const ArdenVariable &var) const
{  return(letter->isAfter(var)); }

ArdenVariable ArdenDefault::isAfter(const ArdenVariable &var) const
{ return(setNull(var)); }

ArdenVariable ArdenVarTime::isAfter(const ArdenVariable &var) const
{ return(operator>(var)); }

ArdenVariable ArdenVarList::isAfter(const ArdenVariable &var) const
{ return(listOperator(var,ArdenVariable::isAfter)); }
```

Fig. 3. C++ implementation of the Arden **IS AFTER** operator.

operations for all Arden variables. An ArdenVarList variable has no associated primary time, but each individual variable in the list has its own primary time.

List variables are implemented by the ArdenVarList class. All the Arden variable operations are implemented by the ArdenVarList class. ArdenVarList contains several support methods (e.g. listOperator) that are used to support list processing. For example, the listOperator member function iterates through the list of Arden variables and calls a function supplied by the caller.

MEDICAL LOGIC MODULE IMPLEMENTATION

Another set of classes were developed to provide the implementation framework for medical logic modules (MLMs). A set of classes implements the textual, textual list and coded slots of a MLM. Objects of these classes comprise the private data for the MLM class. The MLM structured slots are translated into member function implementations of the MLM class by using the pseudo-Arden intermediate language. Mostly, this consists of Arden variables and preprocessor assists that result in a nearly a one-for-one translation.

Each MLM is translated into a separate C++ source file, compiled, and built into shared libraries. These shared libraries can be dynamically loaded, executed, and unloaded by an application using the MLM class as its interface to the logic contained in the shared library. Figure 4 shows a fragment of a logic slot from a MLM used by Columbia-Presbyterian Medical Center. Figure 5 shows the C++ translation for this fragment. These two figures illustrate how the Arden variable classes, when combined with preprocessor assists, can be used to translate an MLM into a compilable C++ program.

There are some restrictions, however, because all of our Arden operators accept only references to ArdenVariable objects as parameters. Expressions like *albumin < 4.0* from Fig. 5 are permitted through an implicit C++ type conversion. The ArdenVariable(const double&) and the ArdenVariable(char*) constructors were added to convert double values and character strings, respectively, to ArdenVariable objects.

```
logic:
  if creatinine is not number and sodium is not number then
    conclude false;
  endif;

  anion_gap := sodium - (chloride + bicarbonate);
  osmolality := 2*sodium + glucose/18 + BUN/2.8;
  BUN_creatinine := BUN/creatinine;
  if albumin < 4.0 then
    corrected_ca := calcium + 0.8*(4.0-albumin);
  elseif albumin > 5.0 then
    corrected_ca := calcium - 0.8*(albumin-5.0);
  else
    corrected_ca := calcium;
  endif;
  ...
  conclude true;
  ;;
```

Fig. 4. Partial MLM logic slot sample source code.

```
Mlm::BoolValue Mlm::runLogic() const
{
  IF(!creatinine.isNumber() && !sodium.isNumber()) THEN
    return(False);
  ENDIF

  anion_gap = sodium - (cloride + biocarbonate);
  osmolality = sodium * 2 + glucose/18 + BUN/2.8;
  BUN_creatinine = BUN/creatinine;
  IF(albumin < 4.0) THEN
    corrected_ca = calcium + (albumin - 4.0) * 0.8;
  ELSEIF(albumin > 5.0) THEN
    corrected_ca = calcium - (albumin - 5.0) * 0.8;
  ELSE
    corrected_ca = calcium;
  ENDIF
  ...
  return(True);
}
```

Fig. 5. Translation of partial MLM logic slot from Fig. 4.

The C++ compiler converts the above expression to *albumin<ArdenVariable(4.0)* because the < operator requires an ArdenVariable parameter. Unfortunately, this restricts C++ expressions to be of the form *ArdenVar .operator. literal.* Expressions such as *1 − ArdenVar* must be reorganized before they can be compiled. In this example, the expression must be changed to *-ArdenVar + 1.*

SUMMARY

Our approach demonstrates that by judiciously using C++ classes, Arden Syntax can be translated into C++ with little necessary conversion. Abstracting MLMs into a class framework further simplifies the execution of Arden Syntax in a C++ environment. We believe that overall complexity and scale of the implementation is less than that of other approaches (even after including the effort required to create the required supporting class libraries). A byproduct of this approach is that developers can use C++ directly to express some forms of clinical medical logic. Conceivably, the framework could be used to combine medical logic with other application logic that is more easily expressed in C/C++.

REFERENCES

1. American Society for Testing and Materials, Standard specification for defining and sharing modular health knowledge bases (Arden Syntax for medical logic modules), *1992 Annual Book of ASTM Standards*, Vol. 14.01, pp.539-587. American Society for Testing and Materials, Philadelphia (1992).
2. G. Hripcsak, J. J. Cimino, S. B. Johnson and P. D. Clayton. The Columbia-Presbyterian Medical Center decision support system as a model for implementing the Arden Syntax, *Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care*, pp. 248–252. McGraw-Hill, New York (1991).
3. B. G. Johansson and O. B. Wigertz, An object-oriented approach to interpret medical knowledge based on the arden syntax, *Proceedings of the Sixteenth Annual Symposium on Computer Applications in Medical Care*, M. E. Frisse, Ed. McGraw-Hill, New York (1993).
4. J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, p. 133. Addison-Wesley, Reading, MA (1992).

About the Author—RONALD A. KUHN is a computer scientist with IBM Corporation. He has designed and developed a variety of applications in the health area such as digital dictation and transcription systems, Radiology PACS (medical imaging and image conveyance), and medical records imaging systems. Currently, he is developing a Clinical Decision Support System that uses Arden Syntax for its knowledge representation language. He received his B.S. and M.S. degrees in Electrical Engineering from the University of Pittsburgh in 1984 and North Carolina State University in 1990, respectively. He can be reached at kuhnr@vnet.ibm.com.

About the Author—ROGER S. REIDER is a system architect with IBM Corporation. He has been designing and developing information systems in the areas of radiology image display and communication (a.k.a. PACS), and medical records imaging systems. His current assignment is designing a set of facilities that can be used to construct a clinical information system with decision support functions. He graduated from the University of Maryland with a B.S. degree in Information Systems Management in 1976. He can be reached at reiderr@vnet.ibm.com.