AN OBJECT-ORIENTED LANGUAGE FOR THE REPORT SPECIFICATION INTERFACE TO A CASE REPOSITORY

By

MANOHAR S.V. RAO

Bachelor of Engineering

University of Mysore

Mysore, India

1988

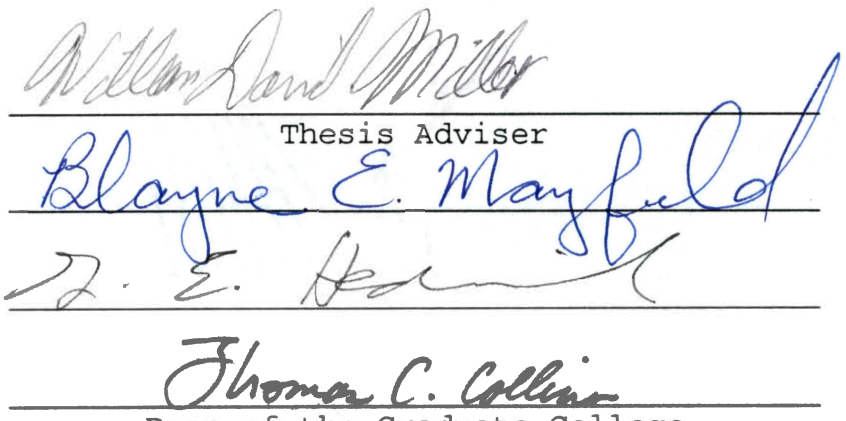
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1992

AN OBJECT-ORIENTED LANGUAGE FOR THE

REPORT SPECIFICATION INTERFACE

TO A CASE REPOSITORY

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

ACKNOWLEDGEMENTS

I am very grateful to Dr. Miller for all the help he has given me in the successful completion of this work. His suggestions for the improvement of this document have made it what it is today. Also, I am very thankful to Dr. Mayfield and Dr. Hedrick for being extremely obliging and helpful and for having put up with my innumerable phone calls in spite of their busy schedules.

A special note of thanks is needed for my collegue at LBMS, Leila Freijy, without whose help, my navigation through the rough waters of word processing would have been a disaster. Similarly, I should thank Gopal Kulkarni and Sridhar Chandrashekar for letting me draw on their experience in defending theses. Also, I have to thank LBMS for allowing me to do this project. In particular, my gratitude is due Yuzo Yamamoto for his technical as well as philosophical guidance.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Object-oriented Programming

Object-oriented programming involves the use of data abstraction. The approach is data-driven, in the sense that the whole process starts with the classification of the data in question into classes (of objects). This classification is based as much on what the objects do as on what they are. The definition of each object consists of both the data elements and the messages that the user of the object can use to access the object. This approach gives us a flexibility not found in traditional programming, because it provides us the desirable qualities of encapsulation, reusability and inheritance. Also, since every function is evaluated on the basis of the object it is referring to, we also obtain the useful quality of polymorphism. This lifts the restriction, usually placed on conventional language users, of having to have unique procedure names. Also, much of the built-in functionality of object-oriented languages is user-configurable.

The CASE Repository

CASE - Computer-Aided Software Engineering - is a combination of software tools and structured software development methodologies [McClure 1989]. A CASE repository is the representation, in data, of all facts about the system under development. This representation is in a form which is independent of its mode of entry or subsequent analysis and reporting. The repository is designed to be accessible to more than one user.

The repository used in this project is based on the Object, Property, Role, Relationship (OPRR) meta model. A meta model is the "database schema" for a CASE repository. It describes the structure and meaning of information that can be stored in the repository and shared among CASE tools [Welke 1989].

This is an object-oriented database system, with the notion of classes and instances implemented in it. The meta model (known as the meta database) is the class definition for the repository. It contains information about the types of objects that will be used in the system based on this model. Inheritance and other interactions between object types are modeled by relationships between objects in the model. An instance database will always comply with the specified meta model and contains instances of the types of

objects and relationships specified. No dynamic changes to the meta model are allowed in this system.

General Statement of the Problem

Given that the OPRR repository is the central place where all the information related to the CASE system resides, the next logical requirement would be some mechanism of database access and report facility. This report facility should be flexible enough to provide the user of the CASE system with easy ways of generating any kind of report that she/he desires, and should be capable of providing the user with different ways of looking at her/his data. Thus, the reporting system should allow the user to define and examine different combinations of the complex relationships existing in the database.

Since the repository itself is object-oriented, the reporting facility should be object-oriented so as to provide the user the conveniences of reusability and inheritance. This means that the user should not only be able to define complex combinations, but should be able to reuse them, and to build upon them, to obtain even more detailed combinations, whenever necessary. Since the reporting facility is object-oriented, it makes it easier to define combinations of data.

Thus, there is the need for an object-oriented reporting language, closely connected to the OPRR repository, which provides the user of the repository with constructs to access the repository, generate reports of any kind desired and of any combination of data desired, and to print out these reports in the format desired.

Objective and Scope of the Study

This study is part of the project that has been started at Learmonth and Burchett Management Systems Incorporated, Houston, Texas (LBMS). The aim of this project is to fulfill the reporting needs of LBMS' database management system Information Manager.

The objective of this study is to design and implement an object-oriented language, RSI2 (Report Specification Interface 2), which gives the user the ability to access and generate reports of the data residing in an OPRR repository. Though most of the constructs in this language will be implemented to suit the OPRR repository, the objective is to implement a language which has the object-oriented qualities of reusability, encapsulation, and inheritance, in such a way that this language itself can be reused in a context other than the OPRR repository without doing a total redesign.

Using this language, the user will be able to define classes consisting of data structures in the repository, or classes that help format the reports that are generated. The user will be able to reuse these classes, or to extend them into new inherited classes, which can provide a new view of the repository.

CHAPTER II

CONCEPTS

Object-oriented Design and Programming

Object-oriented design is the method which leads to software architectures based on a heirarchy of objects. Each system or subsystem is an object in itself, in turn made of objects. Some of the characteristics of object-oriented systems are:

They are modularized on the basis of their data structures. This means that each data structure (called an object), has a behavior associated with it.

Object classes are similar to abstract data types. An abstract data type consists of data with a set of procedures (often called methods) to manipulate them, and the behavior of this data type is limited to the functionality of these methods.

The memory used by deleted objects is reclaimed by the language system, without the programmer having to do anything about it.

Individual objects are grouped into classes. The class definition acts as a factory to produce these object instances. Each instance of a given class differs only in the value of its data, while the behavior remains the same.

New classes of objects can be built by extensions of existing class definitions through the process of inheritance. The subclass (the inheriting class) can reuse all the data and methods of the superclass.

An operation or method is applicable to more than one class of objects through the principle of polymorphism. Thus, we can have more than one message with the same name; the system decides which object the message is referring to at run-time.

The CASE Repository

A repository contains all relevant information about how information systems are constructed and function, be they in planning, under development, or in operation. The repository holds it in a consistent, complete form, independent of the mode of entry, modification, or subsequent use [Welke, 1989]. This information is generally called specifications. The advantages of having a single,

consistent source of system specification information are integrity and ease of use.

The OPRR repository is object-oriented in nature. As mentioned before, there is a meta model or schema which describes the data structure. This would correspond to the classes that we use in an object-oriented language. In addition, there is the instance database, which stores the actual information, conforming to the structure laid down in the meta model or meta database. This instance database corresponds to the instances of the classes in the object-oriented language.

The OPRR system

The repository used here is based on the Objects, Properties, Roles, Relationships model. This model requires a "meta schema" translation and/or programming procedures to represent and manage these schemata. In this model, a target database consists of instances of objects, properties, relationships, and roles. The advantage of the OPRR model is that it allows for many-to-many multipart relationships between objects. The relationships can have properties. Problems associated with the type of data in a relationship, and the direction of flow, are solved in the

OPRR model by having roles in the relationship. The objects participate in a relationship in different roles.

The OPRR system has an application programming interface, which provides a mechanism for an application program to manipulate the "target" database containing instances of objects, relationships, roles and properties whose definition is stored in the "meta" database. This interface is built using object-oriented concepts. Thus, instead of using pointers, data structures, codes, database keys, and other identifying mechanisms, the uniform concept of "object" is used to represent real-life entities. The advantage of this approach is that it isolates the application program from the internal data structure of the system. Each object is identified by a tag called an IDO. Most of the interface functions receive and return IDOs.

In terms of object-oriented programming, there are four different classes of objects that OPRR manipulates: value class, database class, meta class, and list class.

"Definition" objects are used to define OPRR item types and values the application program wants to create or retrieve.

"Database" objects represent instances of objects, properties, relationships, and roles that exist in the

target database. These objects are sometimes called "instances", because they refer to instances of OPRR entities that actually exist in the target database. Specific database objects are referred to as database objects, relationship database objects, role database objects, and property database objects.

"Meta" objects represent object, property, relationship, and role types whose description is stored in the meta database. Meta objects are created using the meta database access functions. Specific meta objects are referred to as object meta objects, property meta objects, relationship meta objects, and role meta objects. Sometimes, meta objects are called OPRR types, or simply types, because they refer to the types of entities that the target database stores.

"List" objects are collections of objects. Any type of OOP objects including list objects may be a member of a list.

"Placeholder" objects are a general enumeration mechanism for a collection. The placeholder object is used to visit each member of a collection.

User-defined objects are objects that are defined by the user. The user of the programming interface, may create new types of objects, and use them in any way desired.

CHAPTER III

THE LANGUAGE SPECIFICATION

Program Structure

An RSI2 program is a sequential set of messages. The message is the only method of execution in RSI2. A message is sent either to the system object - the RSI2 system - or to an instance of a user-defined class. In the first case, the message is a reserved message provided by the RSI2 system, whereas in the second case, the message should be a predefined method of the class in question.

System Messages

These are built-in messages sent to the system object. The RSI2 system object is the global entity that is responsible for executing the system messages, which are predefined and finite in number. The system object is instantiated upon startup of the program, exists as long as there are messages to be executed in the program, and is destroyed only when the program terminates. RSI2 system messages range from control messages like if and while, to

class declaration messages. Appendix C lists all the system messages that are built into RSI2.

User-defined Messages

These are messages recognized by instances of user-defined classes. These messages can only be executed if they have been defined in the class declaration. A typical RSI2 program will be a mixture of system and user-defined messages.

Classes

The most important feature of RSI2 is the ability to define classes, which encapsulate data and the behavior allowed of them.

Declaration

The declaration of a user-defined class in RSI2 consists of three parts:

1. The superclass from which the new class inherits methods and data. This section is optional.

2. The instance variables of the class. This is the data part of the class, and is optional. These variables are only accessible to the methods defined for the class, and are totally hidden from any other class.

3. The methods for the class. These define the behavior of the class, and are the only way the external world can interact with the object of this class. The methods of a class have full access to the data of the class.

Instantiation and Use

For any class to be used, it has to be instantiated. A specific message to the system object will do this. Instances of a class are kept in variables. Use of these instances is only by way of the public messages defined for the class.

Variables

Variables in RSI2 are typeless. A variable can be used to represent an instance of any available class. There are three kinds of variables in RSI2:

Instance Variables

These constitute the data part of a class. They are instances of any available class, and their lifetime is the lifetime of the instance of the class that they are part of. They can be accessed only by the methods of the class they are instance variables of. The instance variables are stored in the object memory.

Temporary Variables

These are variables declared in the methods of a class. They exist only within the scope of the execution of the method and are destroyed when the method returns to its caller. The temporary variables are maintained on the stack which is used by the method that is being executed by the interpreter.

Global Variables

All variables that are neither instance nor temporary variables are global variables. Their lifetime is the lifetime of the program using them, and their scope is the whole program itself. Unlike the previous two kinds of variables, they need not be explicitly declared. When the

interpreter encounters an identifier, it will consider it a global variable if it is neither in the list of temporary variables, nor in the list of instance variables for the class whose method is being executed. This will then become an element in the list of global variables maintained by RSI2 for the program.

Repository Access

The most important aspect of RSI2 is the access to the repository on which the reports are to be produced using the language. The repository has three kinds of entities:

Objects

Objects in the repository can be accessed based on a query. The query is based on an identifying property of the object, or on a relationship that it is involved in, or on a role that it is involved in, in a relationship.

Relationships

Relationships connect objects in the repository. Access to relationships in the database through RSI2 is

again based on a query. The query is based on a property of the relationship, or on one or more objects involved in the relationship.

Properties

Properties in the repository are attributes of either objects or relationships. Again, the query-based access of the property is by object or relationship.

Software specifications for RSI2

Following are the specifications for RSI2:

1. Language of implementation:	C++
2. Hardware:	PC (80x86 based)
3. Operating Systems:	OS/2

CHAPTER IV

THE DESIGN

The RSI2 Processor

The RSI2 processor software can be divided into two parts: the compiler and the run-time environment. The RSI2 compiler reads the RSI2 source statements and generates binary interpretive code. The RSI2 run-time environment includes the RSI2 interpreter.

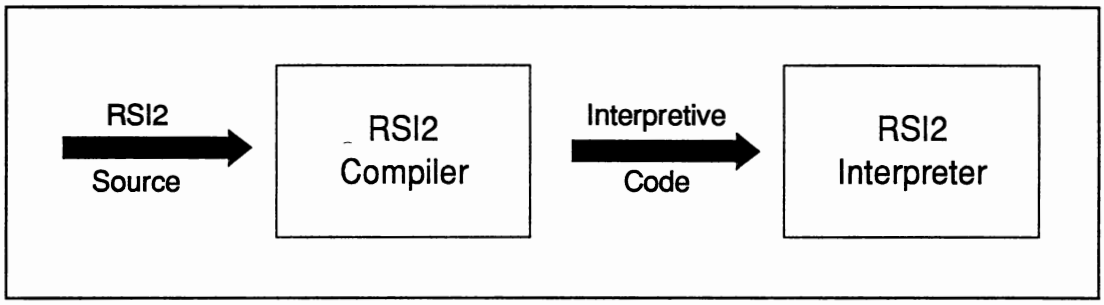


Figure 1. Overview of the RSI2 Processor

Figure 1 shows the data flow for generation of binary interpretive code. The RSI2 compiler generates the interpretive code, which is read by the RSI2 interpreter and

executed. The RSI2 interpreter must be present to execute the RSI2 program. Since the compiler and the interpreter are separate, the binary interpretive code can be distributed without the RSI2 source code, if desired.

The Compiler

Figure 2 shows the overall structure of the RSI2 compiler. The lexical analyzer generates tokens for the parser. The parser consists of the interface to YACC and the routines to build the internal representation of the input language (the parse tree). The code generator uses the parse tree to generate binary interpretive code for the source code.

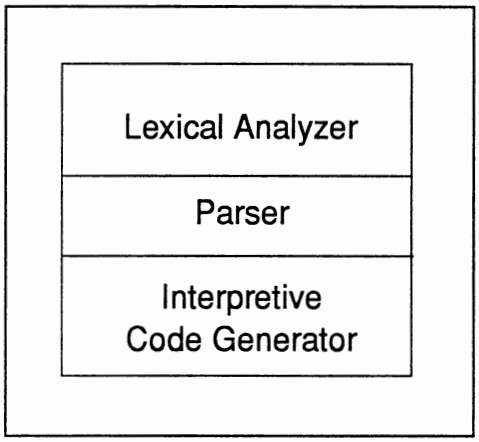


Figure 2. Overview of the RSI2 Compiler.

The compiler compiles one RSI2 message at a time. The messages are separated by the separator character, which is used by the YACC parser and the lexical analyzer to distinguish between messages. Thus at any time, there is only one parse tree in existence, which is for the current message. The working of the compiler can be depicted by the following algorithm, written in pseudocode:

```
RSI2Compiler ()
     {
     get a message from the source file;
     while (there are more messages in the source file)
          {
          parse it into tokens;
          send each token to yacc;
          construct a parse tree for the set of tokens;
          generate binary code for this parse tree;
          get next message from source file;
          }
     }
```

The Lexical Analyzer

The lexical analyzer is the module that deals with the RSI2 source and generates tokens of lexical information for the parser. In RSI2, each message makes up a logical line. So, it is the job of the lexical analyzer to break the source code physical lines into logical lines, and divide them into tokens. The lexical analyzer and the parser agree on a predefined set of types of tokens, and the principal job of the lexical analyzer is to discern the type of each token it generates.

In RSI2, as in any language, there is a set of reserved words, which cannot be used by the user in any sense other than the one in which they have been designed. Each of these reserved words makes up a type of token. In addition to these, there are standard types of tokens, for identifier names, literal strings, etc. Thus the RSI2 lexical analyzer, after removing all nontokens like semicolons, commas, etc., provides the parser with tokens, whenever requested.

The parser asks the lexical analyzer for one token at a time. Thus, the lexical analyzer has also the responsibility of remembering where in the source code the last token was obtained. Using this information, the lexical analyzer gets to the next token by reading in the next section of the source code. Thus, the lexical analyzer is responsible for all the string manipulations that need to be done on the RSI2 source to convert it into an intermediate form - a stream of parser-recognizable tokens - which will be analyzed for correctness by the parser.

The Parser

The parser is also called the syntactic analyzer. This is because the parser verifies the syntax of the source code. Though the parser does not read the source code

directly, it gets one token at a time from the lexical analyzer. The parser has prior knowledge of a set of syntactic rules, known as the grammar of the language, which it uses to check the syntax of the source. Figure 3 shows the input and output of the parser.

In RSI2, the grammar is specified using YACC, which generates a function containing the rule-checking code to be used by the RSI2 parser. Every time the YACC function recognizes a rule, it calls the RSI2 parser for appropriate action to be taken. The RSI2 parser then generates a node, for the token completing the rule, in the parse tree generated for the current message. The YACC grammar for RSI2 is included in Appendix A.

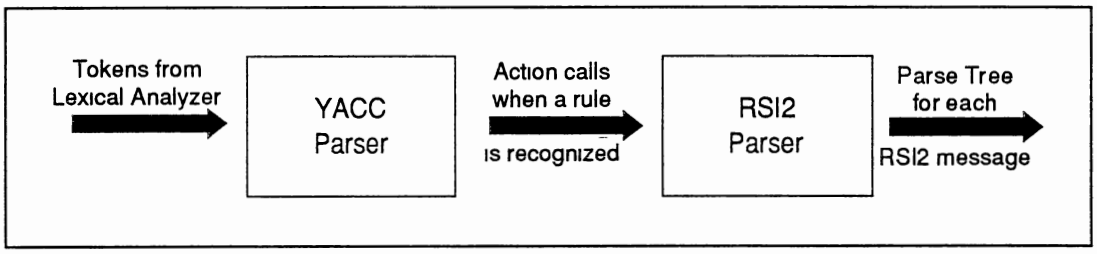


Figure 3. Input/Output for the RSI2 Parser.

In RSI2, the parser generates one parse tree for each message. There is a set of nodes which serve as the header of each parse tree. According to the header token, the

parser generates the corresponding predefined parse tree for the message. Since this is a tree structure, there can be nested parse trees, which gives us the ability to represent complicated expressions, and control structures like the while and the if statements. The predefined set of parse tree formats is also known by the code generator, which always starts from the header node to construct the interpretive code for a message. The parse tree structure for each kind of message in the source code is given in Appendix B.

The Code Generator

After a message is parsed, and found to be syntactically correct, a parse tree is built for that message. The header node for this parse tree is given as input to the code generator. The code generator traverses through the parse tree to generate the interpretive binary code for all the actions taking place in the message. The code generator has prior knowledge of the instruction set of the language, and its main job is to convert the parse tree representation of the code into linear interpretive code. Figure 4 shows the input and output flow for the code generator.

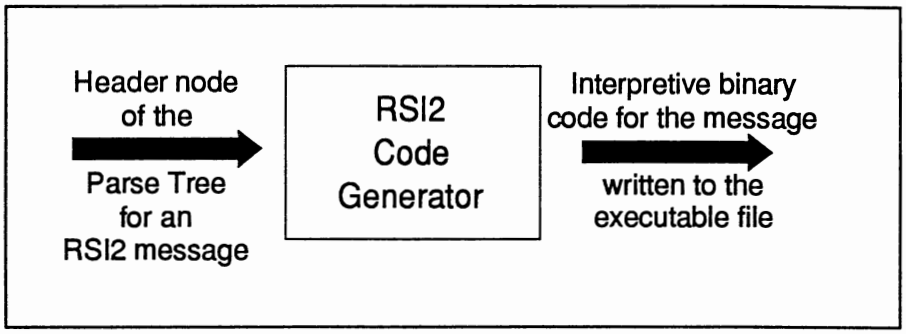


Figure 4. Input/Output for the RSI2 Code Generator

The code generator has methods to deal with any kind of legal message that can be written in an RSI2 program. The methods range from ones dealing with individual nodes, to ones dealing with the message tree as a whole. As mentioned in the previous section, there is a finite set of predefined header node types, that both the parser and the code generator know. Based on the type of the header node input to the code generator, the code generator traverses the parse tree in the appropriate manner to get the information necessary to generate the binary code.

The RSI2 Run-time Environment

Figure 5 shows the run-time environment of RSI2. The run-time support includes implementation of facilities such as handling user-defined temporary variables, and the run-time stack and also performs the role of the linker and loader, and potentially such services as debugger, and

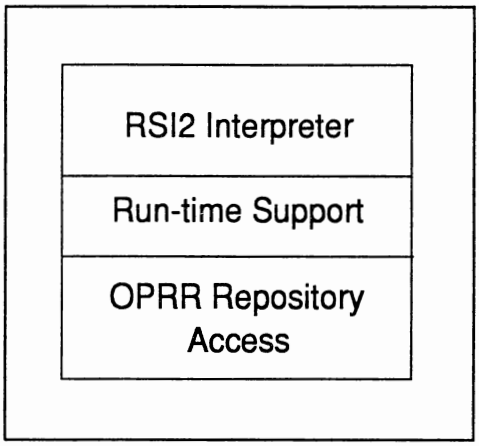


Figure 5. Run-time environment of RSI2

object code librarian. It also includes the handling of RSI2 built-in classes, and interfacing with the repository. The interpreter contains the code that implements the virtual machine.

The Instruction Set

Since RSI2 is an interpretive system, there is the need for an intermediate "language", which the virtual machine "executes". The word "intermediate" refers to the fact that the code is between the very high level class description and the low level language in which the machine is actually operating. There are several reasons for this type of representation. One is compactness; the internal representation of a class description can be much smaller than the character representation used by the creator of the

class. A second reason is efficiency; by translating the class description once into an intermediate representation and thereafter using the internal form, we avoid having to reparse the class description each time a method is invoked. This "language" shall be called the instruction set of the RSI2 system, since it is nothing but a finite set of instructions with different data formats, recognized by the RSI2 interpreter.

In RSI2, all the actions that need to be performed by the interpreter fall into the following categories:

1. Read/Write instance variables.

2. Read/Write temporary and global variables.

3. Read arguments.

4. Read literals.

5. Return expression from a method.

6. Send messages.

7. Perform a primitive operation.

Any RSI2 message (in source code form) can be translated into a subset of this instruction set. In the case of messages that are not built in (calls to methods of

user-defined classes), the instruction to send a message is used with the appropriate method index for the class.

The Run-time Support

The RSI2 run-time support includes the linker/loader and the driver. It also includes managing the operation of built-in classes.

The Linker/Loader. The function of the linker in RSI2 is to link external class libraries to the current program, which will enable us to reuse classes already compiled into libraries. The linker mainly builds up a list of files that are specified by the user as the class libraries, after checking for their validity as class libraries. When the interpreter tries to execute a message to a user-defined class, it first searches for the instructions for that method in the current file, and then looks for that class in the list of files built by the linker. The loader does the job of loading the instructions from the executable file into memory. It also loads class libraries into memory.

The Driver. The RSI2 driver is the module that starts execution of the program. This module is responsible for:

1. maintaining the object memory and class dictionary. Any other module that needs to update/access the system stack will do so using the driver.

2. creating an instance of the interpreter. This instance of the interpreter will be responsible for starting the actual execution of the program.

The Interpreter

The RSI2 intrepreter executes one instruction at a time. It has at its disposal two modules: one for executing system instructions, and one for executing user-defined messages. It is the job of the interpreter to decide the type of the instruction, and do the necessary groundwork, like searching for the receiver of a message, before passing control over to the appropriate module.

The Repository Access

There are four kinds of database objects that can be retrieved from the repository. They are the objects, properties, relationships, and roles. Thus, in the RSI2 language, there are four specific system messages for each of the above retrievals. OPR, the repository access

subsystem, provides us with methods of retrieval of the basic kinds of database objects. The RSI2 retrieval is designed to use these public methods in varying, and possibly complicated, combinations, so that complicated queries can be built in the RSI2 code for their retrieval.

The memory for a database object in the repository is never allocated directly. Each object maintains a counter that indicates the number of times that particular object is being referenced. The reference count must be incremented when a database object is given to an independent programming unit, which in turn decrements the reference count when it no longer needs the database object.

Each OPRR public method definition specifies whether an object returned needs to be decremented by the user or not. The terms "borrowed copy", and "own copy", are used in this context. A borrowed copy of a database object is a copy that is not owned by the accessor of the object. The reference count is not incremented before the database object is obtained. An own copy, on the other hand, is the user's own copy of the database object. Since the reference count of the database object has been incremented for use, the object will not be discarded from memory. The accessor of the database object is responsible for decrementing the reference count of the database object.

Each database object of type Object and Relationship has a permanently assigned unique identification number (UID). The UID is not reused again in the same database. The actual value of the UID is assigned by OPRR and is not under the control of the user program. The UID itself, however, can be retrieved, or an object corresponding to a given UID can be retrieved. Only Objects and Relationships have unique IDs. A common set of unique IDs is used for both Objects and Relationships.

In OPRR, an identifying property of an object need not be unique over the whole database, but only within the meta type of the objects that it applies to. Furthermore, uniqueness is only needed within what is called a scoping relationship. In a scoping relationship, two roles are important: the scoping role and the scoped role. A database object which is scoped cannot exist before the scoping relationship is created. Also the scoped object exists only within the scope of the scoping object, which means that the scoped object is deleted when the scoping object is deleted. This gives rise to the need for special processing within RSI2 to keep track of the scoping heirarchy.

RSI2 supports the simultaneous use of only one meta database although several target databases may be open at a time. This restriction is imposed on the system by the OPRR subsystem. Each operation relating to an object, property,

relationship or role specifies the target database the operation applies to.

There are specific methods to access the meta schema, and specific methods to access the target database. The RSI2 compiler uses only the meta database access functions to check to see that the types specified by the user exist in the schema, whereas the run-time environment uses both the meta database access, and target database access methods.

CHAPTER V

THE IMPLEMENTATION

The Compiler

This section deals with the C++ class structure used to implement the compiler based on the design given earlier. Figure 6 shows the class structure of the compiler. There are three distinct kinds of classes that are used in the implementation:

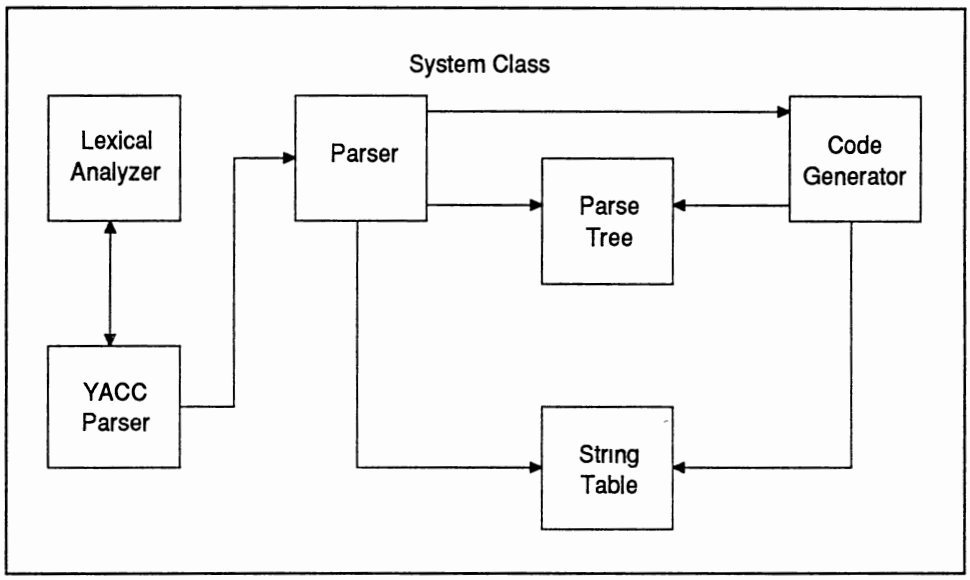


Figure 6. Class Structure for the RSI2 Compiler

The System Class

This is the global class whose scope is the entire compiler itself. This can be called the Compiler. As shown in figure 6, this class encompasses all the component and accessory classes. It is basically the manager of the interaction between the various classes. The following are the functions of the Compiler class:

1. Managing the interaction between the YACC parse function and the Lexical Analyzer object. This mainly involves giving calls to the lexical analyzer to get the next token whenever the YACC parse function needs it.

2. Managing the interaction between the parser and the code generator. This involves storing the header node of the parse tree representing the current message being compiled, so that the CodeGenerator object can use that as the input for generating binary output code for the current message.

3. Instantiating and destroying all the component classes.

4. Maintaining all global variables.

5. Command line processing when the compiler is invoked by the user.

The Component Classes

These are the main classes that are used in the implementation and they follow the design explained in the previous chapter:

The LexicalAnalyzer Class. This class provides the implementation of the lexical analyzer. The instance variables of this class serve the following purposes:

1. A handle to the input file from which the source is being read.

2. The current line number in the input source file.

3. The current position in the current line in the input source file.

4. Several flags to indicate whether the debug switch is on, the warning level, whether the source is to be listed when compiling, etc.

5. Several buffers for all the string manipulation that needs to be done.

The public methods serve the following purposes:

1. Initializing the object, which includes setting the input file handle, and initializing all the flags.

2. Getting the next token in the input stream.

3. Handling an error returned by YACC.

The Parser Class. This class works in conjunction with the ParseTree class, and mainly contains actions to be performed whenever the YACC parser indicates the completion of a rule. The public methods serve the following purposes:

1. Initialization of the object.

2. Actions for the completion without errors of each rule by the YACC parser.

The ParseTree Class. This class contains the implementation of the parse tree node. It works in conjunction with the StringTable object, since that is where all the literal data are kept. There are two kinds of nodes in RSI2 - the operator node and the data node. Each node,

irrespective of its kind, contains type information, number of child nodes, and an index to the brother node (next sibling). While the operator node contains pointers to the first and last child nodes, the data node contains an index into the string table for string data, and actually keeps literal integer data in the node itself. The instance variables of this class serve the following purposes:

1. A node which is the union of the two kinds of nodes explained above.

2. The header of this parse tree.

3. The next available spot in the parse tree for the insertion of a new node.

The public methods serve the following purposes:

1. Initialization of the object, which includes resetting the header pointer, and the next node pointer.

2. Creation and deletion of nodes.

3. Insertion and removal of nodes.

4. Access and modification of data contained in nodes.

The CodeGenerator Class. This class also works in conjunction with the ParseTree class, and mainly contains methods to convert the given parse tree into interpretive binary code. The public methods of this class serve the following purposes:

1. Initialization of the object, which includes storing the header node for the input parse tree.

2. Methods for the emission of binary instructions for the parse tree that is provided as input to the object

The Accessory Classes

These are classes that are not RSI2 specific, but used for various purposes. These need not necessarily be C++ classes, but could be pseudoclasses:

The StringTable Class. This class stores all the literal strings and the identifiers in the source program. This is used by the Parser and ParseTree classes to access the various strings. The string table is basically an array of characters, with an additional array providing the mapping that is needed to separate the individual strings. Thus the string table is a linear arrangement of strings, with the map indicating the type of each string and its

displacement in the table. The instance variables of this class serve the following purposes:

1. The mapping of the individual literals in the table, with information about the type of literal.

2. The handle to the area in memory where the whole string table is stored.

3. Information about the next available spot in the string table.

The public methods serve the following purposes:

1. Initialization of the class, which includes procuring the specified amount of space for the string table.

2. Addition to the table at the next available spot and deletion of a string.

3. Access of strings whose ids are given.

4. Utility functions like comparing two strings in the string table, getting the length of a string in the string table, converting to upper case, etc.

The YACC parser Pseudoclass. This is actually a C function that is generated by the YACC parser, given the grammatical rules for the language. This function works in conjunction with the lexical analyzer object, and in fact, is the only user of the lexical analyzer object other than the system class object. This function is yyparse (). It asks for the next token in the input stream, decides whether this is a valid token, and whether it completes any rules. In case a rule is completed, this function calls the corresponding public method of the parser object.

The Run-time Environment

This section deals with the C++ class structure used to implement the run-time evironment. The class structure is based on the design given in the previous chapter. Figure 7 shows the class structure of the run-time environment. There are three distinct kinds of classes that are used in the implementation:

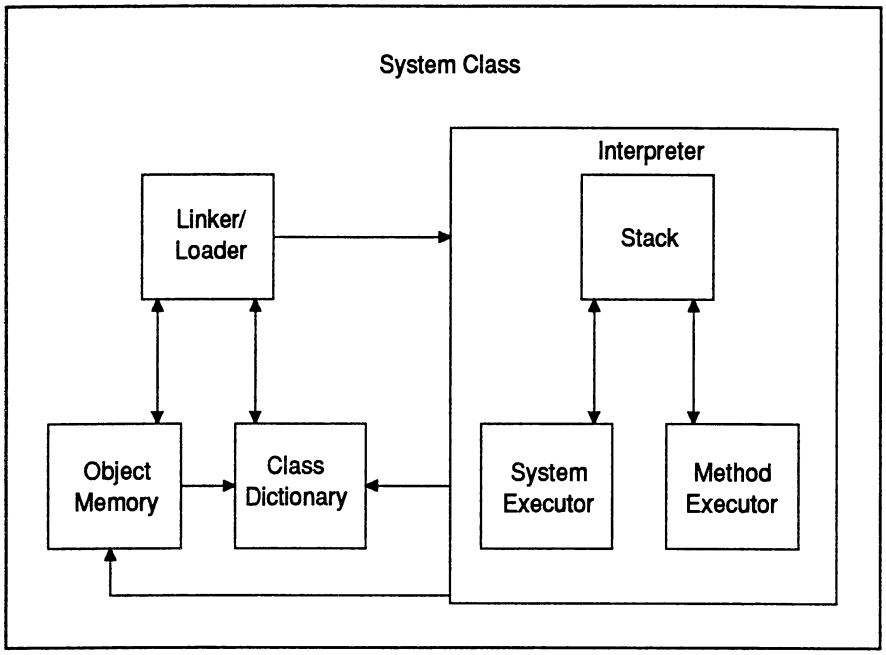


Figure 7. Class Structure for the RSI2 Run-time Environment

The System Class

This is the global class whose scope is the entire run-time environment itself. This can be called the Driver. As shown in figure 7, this class encompasses all the component and accessory classes. It is basically the manager of the interaction between the various classes. The following are the functions of the system class:

1. Managing the interaction between the loader, the object memory and the class dictionary.

2. Initiating the interpreter by filling in its memory with the code for the main program.

3. Instantiating and destroying all the component classes.

4. Maintaining all global variables.

5. Command line processing when the interpreter is invoked by the user.

The Component Classes

These are the main classes that are used in the implementation and they follow the design explained in the previous chapter:

The Linker Class. This class provides the implementation of the linker/loader. The instance variables of this class serve the following purposes:

1. A handle to the input executable file from which the code is being read.

2. The current position in the current line in the input source file.

3. Several buffers for all the string manipulation that needs to be done.

The public methods serve the following purposes:

1. Initializing the object, which includes setting the input file handle, and initializing all the flags.

2. Getting all the classes from all the library files specified on the command line, and adding them to the Driver's global list of library files.

The ObjectMemory Class. This class works in conjunction with most of the other classes. This is a subclass of the Array class since it is just an array of indices and strings. This class keeps all the global variables of the RSI2 program being run. The public methods serve the following purposes:

1. Initialization of the object.

2. Getting the value of the specified variable.

3. Updating the value of a specified variable.

The ClassDictionary Class. This class contains information about all the classes being used in the program.

This includes the file in which the implementation of the class resides and the inheritance structure of each class. The instance variables of this class serve the following purposes:

1. An array of class names.

2. An array of file handles for all the classes.

The public methods serve the following purposes:

1. Initialization of the object.

2. Insertion and removal of classes from the dictionary.

3. Search for a given class.

The Interpreter Class. This class is the main subsystem in the run-time environment. The Driver gives control to this class after the initial linking and loading are done. The instance variables of this class serve the following purposes:

1. An array of instructions of the main program to be executed.

2. A stack to maintain the message call structure.

The public methods of this class serve the following purposes:

1. Initialization of the object.

2. Execution of each instruction.

The SystemExecuter Class. This class is the primitive handler of RSI2. The Interpreter gives control to this class whenever a system message is to be executed. The public methods of this class serve the following purposes:

1. Initialization of the object.

2. Execution of each system instruction.

The MethodExecuter Class. This class is the method handler of RSI2. The Interpreter gives control to this class whenever a method is to be executed. The instance variables of this class serve the following purposes:

1. An array of instructions of the method to be executed.

2. A stack to maintain the temporary variables.

The public methods of this class serve the following purposes:

1. Initialization of the object.

2. Execution of each instruction in the method.

The Accessory Classes

These are classes that are not RSI2 specific, but used for various purposes:

The Array Class. This class is a generic class that includes the implementation of the array in C++. This has been reused from available implementations.

The Stack Class. This is also a generic class that includes the implementation of the stack in C++. This has been reused from available implementations. This is a subclass of the Array class.

CHAPTER VI

COMPARISON WITH RSI1

The following are some of the salient differences between RSI2 and the existing reporting language - RSI1 - that has been partially fulfilling the needs of the OPRR repository:

1. The most important difference is the support provided for object-oriented programming in RSI2. As mentioned earlier, the CASE repository is object-oriented in nature, and the ability to build classes for the different objects in the database is extremely helpful.

2. Relationships in the repository model inheritance, and RSI2 provides for inheritance of classes, which could model the relationships, and provide for extremely user-friendly report specification for relationships.

3. RSI1 does not in any way support reusability of source code; RSI2 gives the user the ability to build class libraries, which can be used in other RSI2 programs.

4. RSI1 is implemented in C, whereas RSI2 is implemented in C++, which allows for a structured design based on classes.

5. In RSI1, access to the repository is list based and is cumbersome to use. Accessing a particular relationship requires getting a list of all relationships answering the given query, and getting the next available relationship until the one needed is obtained. A simple built-in message solves this problem in RSI2.

CHAPTER VII

CONCLUSION AND SUGGESTIONS FOR FUTURE WORK

Conclusions

RSI2 is proposed to be an object-oriented language which provides the report specification interface to an OPRR CASE repository. Using the language, the user can develop classes, and send messages to an instance of a class, called an object. The object-oriented nature of the language provides the user with the advantages of class and method reusability, inheritance and encapsulation. Because the language is object-oriented, new features may be added efficiently by a class library without modifying the language itself. In addition to creating a completely new class library, a new class may be created as a subtype of an existing object, inheriting all the properties and methods, and adding and overriding selected methods.

Future Work

Future versions of RSI2 could have the following enhancements:

1. A window based operating environment. Presently, the only way of user interaction with the RSI2 system is through the command line in OS/2. The system should be made much like the SmallTalk environment, where the user can build classes using intuitive windows, and other graphic user interface tools.

2. Multiple Inheritance. Presently, RSI2 allows only single inheritance. It should allow the construction of classes which inherit data and behavior from more than one class.

3. Making changes to the repository. Presently, RSI2 has only read-only access to the repository. This is sufficient since RSI2 is a reporting facility. This restriction can be removed, and facilities provided to modify the data in the repository.

4. Error Handling using the exception handling mechanism of C++. The present version of C++ used in the implementation does not have the exception handling capability that is part of the C++ specification. This should be used once it is available, since it makes context-sensitive error handling in RSI2 much easier.

5. Interface to popular programming languages. RSI2 could be made to respond to function calls in popular

programming languages like C and C++. This could extend the power of RSI2 to include the desirable features of these languages.

6. C code generation. The code generator of RSI2 now generates interpretive binary code which can only be executed by the RSI2 interpreter. If it is made to generate C or C++ code for a given RSI2 source file, then they could be made into dynamic link libraries or executables which might provide significant performance improvements.

7. More sophisticated list processing. Currently, RSI2 can only generate homogeneous lists of instances of user-defined classes. It could be improved to incorporate bags, sets, and such other commonly known collection types.

8. Optimization. RSI2, currently, does not have any optimization done on the code at all. There is much scope for optimization in code generation.

9. Provision for Class Variables. RSI2 does not have support for class variables global to all instances of the class.

BIBLIOGRAPHY

Actor User's Manual Vol. 1 and 2, The Whitewater Group, 1991.

Aho, A.V., Ullman, J.D., Principles of Compiler Design, Addison Wesley, 1979.

Andersen, J., Reenskaug, T., "Operations on sets in an OODB", OOPS MESSENGER, Vol. 2, No. 4, pp. 26-39, October 1991.

Booch, G., "Object-Oriented Development", IEEE Trans. Software Engg., Vol. SE-12, pp. 211-221, Feb. 1986.

Budd, T., A Little Smalltalk, Addison Wesley, 1987.

Cox, B.J., Object-Oriented Programming, Addison Wesley, 1987.

Goldberg, A., Smalltalk-80 The Interactive Programming Environment, Addison Wesley, 1984.

Goldberg, A.,Robson, D., Smalltalk-80 The Language and its Implementation, Addison Wesley, 1985.

LBMS Information Manager User's Manual, LBMS Inc., 1991.

McClure, C., CASE in Software Automation, Prentice Hall, 1989.

McClure, C., "The CASE Experience", BYTE, pp. 235-246, April 1989.

Meyer, B., Object-oriented Software Construction, Prentice Hall, 1988.

Norman, R.J., Nunamaker Jr., J.F., "CASE Productivity Perceptions of Software Engineering Professionals", Communications of the ACM, Vol. 32, No. 9, pp. 1102-1107, Sept. 1989.

Smalltalk/V Windows Tutorial and Programming Handbook, Digitalk Inc., 1991.

Specification for OPRR Application Programming Interface, Version 2, LBMS Inc., Ann Arbor, 1991.

Welke, R.J., "Meta Systems on Meta Models", CASE Outlook, Vol. 4, pp. 35-43, 1989.

Yamamoto, Y., Personal communications with Yuzo Yamamoto, 1991-92.

APPENDICES

APPENDIX A

RSI2 LANGUAGE GRAMMAR

```
/*
 * File:    rsigramm.y (RSIGRAMMar.Yacc)
 * Purpose: yacc grammar for RSI Language
 * History:
 *  91/09/18 mr  Created
 */

%{
#define INCLUDE_STDARG
#define INCLUDE_STRING
#define INCLUDE_CTYPE
#define INCLUDE_STDIO
#include "machine.h"
#include "rsicomi.hpp"
#include "rsilex.hpp"
#include "rsiparsr.hpp"
extern RSI_Parser* RSI_pclParser;
%}

%union
    {
    char    stid;        /* string table id */
    short   n;           /* a small integer */
    }

%token  NAMETOKEN INTEGERTOKEN FLOATTOKEN STRINGTOKEN ASSIGNTOKEN
%token  NUMBERTOKEN TEXTTOKEN
%token  BEGINBLOCK ENDBLOCK ENDMSG
%token  KWD_CLASS KWD_SUPER KWD_SELF KWD_SUPERCLASS
%token  KWD_START KWD_VAR KWD_METHOD KWD_INSTANCEOF
%token  KWD_WRITE KWD_ADD KWD_ASSIGN KWD_SUBTRACT
%token  KWD_IF KWD_ELSEIF KWD_ELSE KWD_WHILE KWD_RETURN
%token  KWD_LIST KWD_PARAMETER
%token  KWD_OPROBJ KWD_OPRREL KWD_OPRPROP

%nonassoc GT GE LT LE '=' NE
%left   '|' '&'
%left   '~'
%left   '+' '-'
%left   '*' '/'

%type   <n> messageblock message sysmessage objmessage
%type   <n> superclassdecl classdecl instofdecl vardecl methoddecl
%type   <n> classbody methodbody
%type   <n> messagelist varlist
%type   <n> oprobjmsg oprrelmsg oprpropmsg oprquery
%type   <n> oprobjmsgbody oprrelmsgbody
%type   <n> writemsg assignmsg parametermsg
```

```
%type   <n> ifmsg whilemsg returnmsg
%type   <n> elseifblock elseifbody elseblock

%start program
%%

program
            :   KWD_START messageblock
                { RSI_pclParser->doProgram (); }
            |   classdecl
                { RSI_pclParser->doClassLib (); }
              ;

messageblock
            :   BEGINBLOCK messagelist ENDBLOCK
                { ; }

messagelist
            :   /* nothing */
                { ; }
            |   messagelist message
                { ; }
            ;

message
            :   sysmessage
                { ; }
            |   objmessage
                { ; }
            ;

sysmessage
            :   vardecl
                { ; }
            |   classdecl
                { ; }
            |   instofdecl
                { ; }
            |   oprobjmsg
                { ; }
            |   oprrelmsg
                { ; }
            |   oprpropmsg
                { ; }
            |   writemsg
                { ; }
            |   assignmsg
                { ; }
            |   ifmsg
                { ; }
            |   whilemsg
                { ; }
            |   parametermsg
                { ; }
            |   returnmsg
                { ; }
            ;

classdecl
            :   KWD_CLASS NAMETOKEN ':' BEGINBLOCK classbody ENDBLOCK
                { ; }
            ;

classbody
```

```
            :   superclassdecl vardecl methoddecl
                { ; }
            |   vardecl methoddecl
                { ; }
            |   methoddecl
                { ; }
            ;

superclassdecl
            :   KWD_SUPERCLASS '(' NAMETOKEN ')' ENDBLOCK
                { ; }
            ;

vardecl
            :   KWD_VAR '(' varlist ')' ENDMSG
                { RSI_pclParser->doVarDecl (); }
            ;

varlist
            :   NAMETOKEN
                { ; }
            |   varlist ',' NAMETOKEN
                { ; }
            ;

methoddecl
            :   methodbody
                { ; }
            |   methodbody methoddecl
                { ; }
            ;

methodbody
            :   KWD_METHOD NAMETOKEN ':' messageblock
                { ; }
            ;

instofdecl
            :   KWD_INSTANCEOF '(' NAMETOKEN ')' ENDMSG
                { ; }
            |   KWD_INSTANCEOF '(' KWD_LIST ')' ENDMSG
                { ; }
            ;

oprquery
            :   NAMETOKEN GT constvalue
                { ; }
            |   NAMETOKEN GE constvalue
                { ; }
            |   NAMETOKEN LT constvalue
                { ; }
            |   NAMETOKEN LE constvalue
                { ; }
            |   NAMETOKEN '=' constvalue
                { ; }
            |   NAMETOKEN NE constvalue
                { ; }
            ;

oprobjmsg
            :   oprobjmsgbody ENDMSG
                { ; }
            ;
```

```
oprobjmsgbody
            :   KWD_OPROBJ '(' NAMETOKEN ',' oprquery ')'
                { ; }
            |   KWD_OPROBJ '(' NAMETOKEN ')'
                { ; }
            ;

oprrelmsg
            :   oprrelmsgbody ENDMSG
                { ; }
            ;

oprrelmsgbody
            :   KWD_OPRREL '(' NAMETOKEN ',' oprobjmsgbody ')'
                { ; }
            |   KWD_OPRREL '(' NAMETOKEN ',' varlist ')'
                { ; }
            |   KWD_OPRREL '(' NAMETOKEN ')'
                { ; }
            ;

oprpropmsg
            :   KWD_OPRPROP '(' NAMETOKEN ',' oprobjmsgbody ')' ENDMSG
                { ; }
            |   KWD_OPRPROP '(' NAMETOKEN ',' oprrelmsgbody ')' ENDMSG
                { ; }
            |   KWD_OPRPROP '(' NAMETOKEN ',' NAMETOKEN ')' ENDMSG
                { ; }
            ;

objmessage
            :   NAMETOKEN '(' varlist ')' ENDMSG
                { ; }
            |   NAMETOKEN '(' KWD_SELF varlist ')' ENDMSG
                { ; }
            |   NAMETOKEN '(' KWD_SUPER varlist ')' ENDMSG
                { ; }
            ;

writemsg
            :   KWD_WRITE '(' NAMETOKEN ')' ENDMSG
                { RSI_pclParser->doWrite (); }
            ;

assignmsg
            :   NAMETOKEN ASSIGNTOKEN tokensegment ENDMSG
                { ; }
            ;

ifmsg
            :   KWD_IF condition messageblock elseifblock elseblock
                { ; }
            ;

elseifblock
            :   /* nothing */
                { ; }
            |   elseifbody elseifblock
                { ; }
            ;

elseifbody
            :   KWD_ELSEIF condition messageblock
                { ; }
```

```
            ;

elseblock
            :   /* nothing */
                { ; }
            |   KWD_ELSE messageblock
                { ; }
            ;

whilemsg
            :   KWD_WHILE condition messageblock
                { ; }
            ;

returnmsg
            :   KWD_RETURN tokensegment ENDMSG
                { ; }
            ;

parametermsg
            :   KWD_PARAMETER '(' INTEGERTOKEN ')' ENDMSG
                { ; }
            ;

condition
            :   logicalexpr
                { ; }
            ;

logicalexpr
            :   logicalterm
                { ; }
            |   logicalexpr '|' logicalterm
                { ; }
            |   logicalexpr '&' logicalterm
                { ; }
            ;

logicalterm
            :   tokensegment
                { ; }
            |   '~' logicalterm
                { ; }
            |   tokensegment GT tokensegment
                { ; }
            |   tokensegment GE tokensegment
                { ; }
            |   tokensegment LT tokensegment
                { ; }
            |   tokensegment LE tokensegment
                { ; }
            |   tokensegment '=' tokensegment
                { ; }
            |   tokensegment NE tokensegment
                { ; }
            ;

tokensegment
            :   objmessage
                { ; }
            |   expression
                { ; }
            ;
```

```
expression
            :   constvalue
                { ; }
            |   expression '+' expression
                { ; }
            |   expression '-' expression
                { ; }
            |   expression '*' expression
                { ; }
            |   expression '/' expression
                { ; }
            ;

constvalue    /* Constant values */
            :   TEXTTOKEN
            |   NAMETOKEN
            |   STRINGTOKEN
            |   INTEGERTOKEN
            |   NUMBERTOKEN
            ;
```

APPENDIX B

THE PARSE TREE NODE STRUCTURE

The following figures depict the different kinds of parse trees that are constructed by the RSI2 parser. The following are the types allowed:

1. MESSAGE
2. IDENTIFIER
3. VALUE

The value section for the MESSAGE type node above can have these possible values:

START, CLASS, VAR, INSTANCEOF, IF, WHILE, WRITE, RETURN, METHOD, METHODCALL, DBOBJ, DBREL, DBPROP, DBROLE, LIST, +, - , * , /, GT, GE, LT, LE, NE.

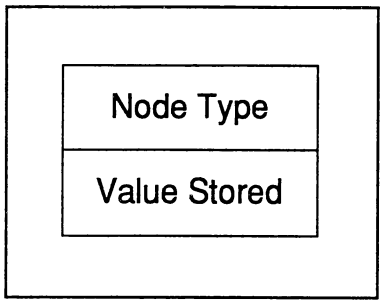


Figure 8. Notation for an RSI2 parse tree node

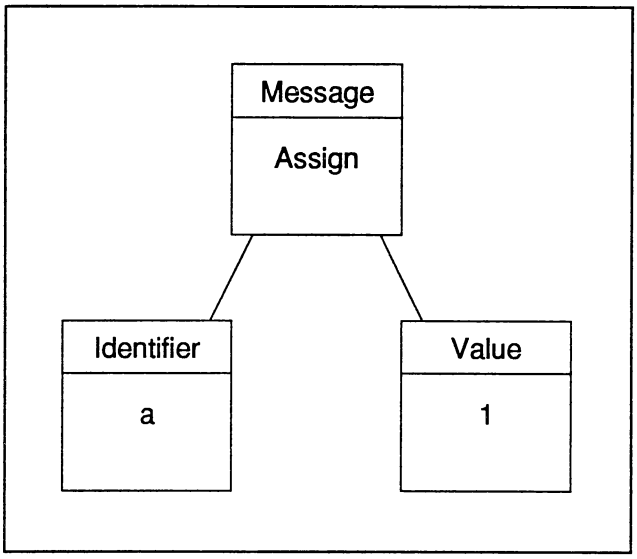


Figure 9. A simple RSI2 statement

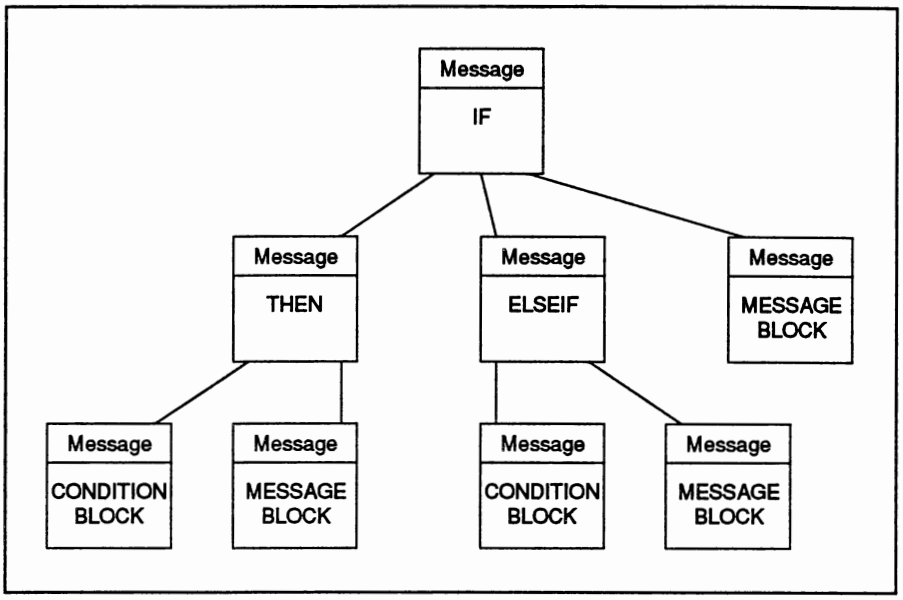


Figure 10. The IF statement

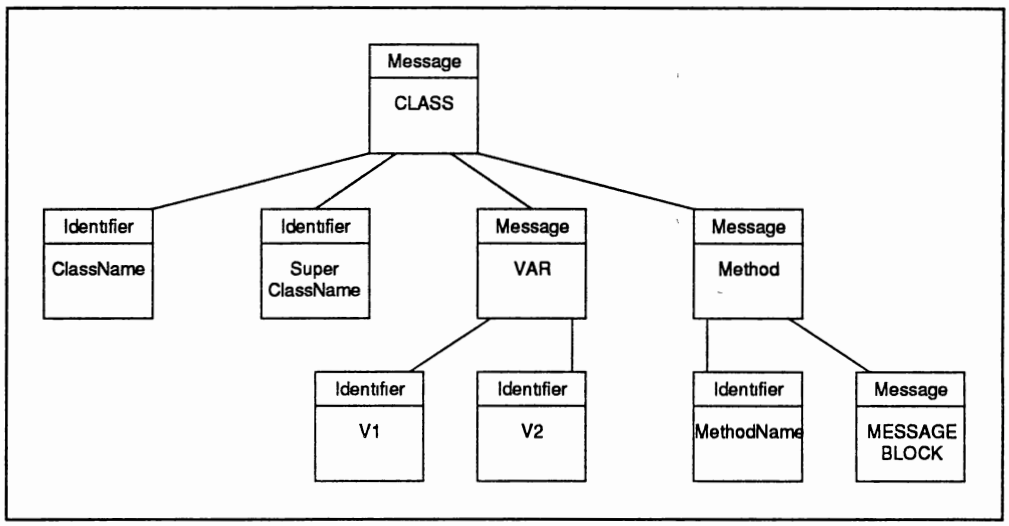


Figure 11. The CLASS Statement

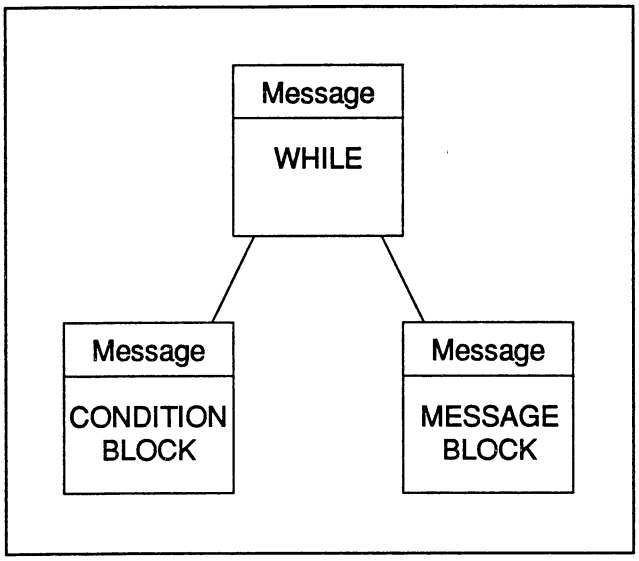


Figure 12. The WHILE Statement

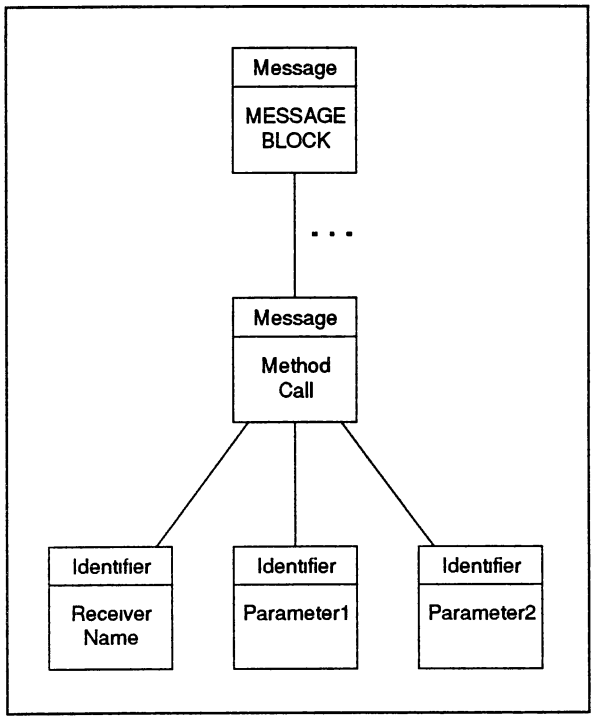


Figure 13. A Message Block

APPENDIX C

USER'S MANUAL

Program Structure

An RSI2 program is made up of messages. The messages are either to instances of user-defined classes, or to the system object. An RSI2 source file can be made to be either an executable, or a library. The compiler decides whether the source is to result in an executable or a library, by looking for a specific keyword in the source. This is the "start" keyword. If this exists in the source, then the resulting binary file will be an executable, and can be executed by the interpreter. On the other hand, if the start keyword does not exist, then the file is assumed to be the source for a class library, and there should be at least one class definition in the file for it to be a valid file.

Messages in an RSI2 source file are separated by semicolons. Semicolons are not needed when a message encloses a block of messages within braces, as in an IF message. The characters /* start a comment in RSI2 that terminates with the characters */. Any characters can be used within a comment.

Messages

Messages in RSI2 are similar to function calls in conventional programming languages. The difference lies in the fact that the first parameter to a user-defined message is always the receiver of the message. Also, RSI2 does not tightly control the number of parameters that are sent to a message. It is upto the user to be careful not to try to use a parameter that was not actually sent to the method. None of the system messages need a receiver, since their identifiers are keywords, and the compiler immediately converts them into messages to the system object. There are two special kinds of receivers, namely self, and super. "self" is used in a user-defined method to give a call to a method that is part of the same class. "super" is a call to a method that is part of the interface of the superclass from which the present class is inheriting its methods.

Since compiling and interpreting are done separately in RSI2, the system object exists in two forms: the compiler system object, and the run-time system object. This distinction is needed because, some of the built-in messages are messages to the compiler system object, whereas the others are to the run-time system object. The messages to the compiler system object are static, whereas the others are dynamic. The messages to the compiler system object are not converted into intermediate binary code for

interpretation. All messages involving allocation of space are to the compiler. The class declaration and the variable declaration are examples of messages to the compiler system object. Run-time messages include messages like while, and if, which do not need permanent memory allocation.

Class Declaration

The following shows the class declaration in RSI2. The superclass and var messages are optional. The compiler does not check to see if the superclass exists or not. The linker does that by searching for that class in all the linked files. Every method definition starts with the "method" keyword, which intimates the compiler that a method definition is being made. The var section in each method is optional.

```
class ClassName:
     {
     superclass(SuperClassName);
     var(instance,variables);

     method method1:
          {
          var(temporary,variables)
          /* statements */
          }
     method method2:
          {
          /* statements */
          }
     }
```

Built-in Classes

The following are the built-in classes which RSI2 uses for its various operations. The system class has exactly one instance in memory during the entire session, and the primitive classes are the interface to the actual machine.

The System Class.

There will always be one instance of this class open during an RSI2 session. This is a virtual class, in the sense that, there is actually no one system class in the C++ implementation, but a combination of several, for the various system messages. But, to the user, this is just one class, which acts as the receiver of all the built-in messages provided by RSI2 to the user.

The Primitive Classes.

The primitive classes of RSI2 are integer, float, character, and string. These are called primitive, because RSI2 itself does not have an implementation for the various operations that can be performed on them. These classes act

as the interface between the RSI2 system and the underlying physical machine.

Constants and Variables

Constants in RSI2 are treated in the same way as in other languages. There are three kinds of constants in RSI2:

Number Constants: These include integer constants and floating point constants. Integer constants are numbers like 0, 18, 1900000, etc., whereas floating point constants are numbers like 1.23, 1.2e10, etc.

Character Constants: A character constant is a character enclosed in single quotes; e.g., 'a', '9'. The backslash \ is used as an escape character, so that special characters get representation.

String Constants: A string constant is a character sequence enclosed in double quotes; e.g., "this is a string". Every string constant is terminated by the null character '\0', with the value 0. The backslash convention for representing nongraphic characters can also be used within a string.

Variables in RSI2 are basically containers of objects. They do not have types. Since they are containers, they can point to any object of any class. There are three kinds of variables in RSI2:

Instance Variables: These are the data part of an object. Each instance of a class has a copy of these. These instance variables are declared in the class declaration as shown before. All instance variables will initially be NULL objects.

Temporary Variables: These are used in methods as temporary containers of the different objects being processed. RSI2 destroys the space held by them, as soon as the the execution of the method is over. Their declaration is similar to that of the instance variables - using the var message to the system.

Global Variables: Any variable that has not been declared using the var message, and that is not a class identifier, is a global variable. The difference between a global variable and a temporary variable is that RSI2 does not destroy the space held by global variables; they exist throughout the scope of the program.

Reserved Messages

The following is the set of all the built-in messages at the disposal of the RSI2 user. Each description also indicates whether the message is to the compiler system object, or the run-time system object.

start

This keyword indicates to the compiler the starting point of execution of the program. The start message can occur only once in a source file and it cannot occur inside any other message or method. An RSI2 executable can only be produced from a source file containing the start message. This is a message to the compiler system object.

class

This keyword does the job of notifying the compiler that a class is being defined. The class declaration as shown earlier follows this word. This message can occur anywhere in the program, except inside a class declaration. If a source file contains only class messages, then the compiler will not produce an executable for the interpreter to execute; it will produce a library file which can be used

by other executables. This is a message to the compiler system object.

var

This serves to notify the compiler that the following identifiers will serve as variables in the program, or class. This is a message to the compiler system object.

instanceof

This notifies the compiler that a given variable is going to be an instance of a given class. This is a message to the run-time system object. The compiler produces interpretive code to do this. None of the primitive classes need instantiation using this message. The following is an example of how this message is used:

```
var(a);  /* declare a as a variable */
a = instanceof(AClass); /* a is an object of AClass */
```

if

This is the control statement and is used as it is used in any other language. This is a message to the run-time system object. An example:

```
if (a = 9)
     {
     /* messages */
     }
elseif (a GT 9)
     {
     /* messages */
     }
elseif (a LT 9)
     {
     /* messages */
     }
else
     {
     /* messages */
     }
```

while

This is similar to the IF message. The loop is executed as long as the condition evaluates to true. This is a message to the run-time system object, for which the compiler produces interpretive binary code. An example:

```
while (IsChar (a))
     {
     /* messages */
     }
```

write

This message is to the run-time system object to write the given string of bytes to the output. The compiler will write the literal string out to the executable file, from where the interpreter picks it up before writing it to the output. An example:

```
write ("Hello");
```

return

This message is used to return a value from a method. The returned value can be a variable, an expression, or even a message to another object. This is a message to the run-time system object. An example:

```
method GetObjectFromDB:
     {
     return (dbobj (PROJECT, REPOSITORY-ID=1);
     }
```

superclass

This message is only used within a class declaration, and notifies the compiler of the name of the class from which the class being declared is inheriting behavior. This is a message to the compiler system object. The compiler writes this name out in the executable file in the class declaration section.

method

This message is also used only in a class declaration. This notifies the compiler that what follows is the source

for a method for the class being declared. This is a message to the compiler system object. As in the case of the superclass message, the compiler writes out the name of the method in the method section of the executable or library file. The compiler also generates binary code for the implementation of the method and places that in the same section in the output file.

super

This is special keyword used only within the implementation of the method to explicitly notify the interpreter that the message being sent is to the superclass.

self

This is similar to the super keyword. This will be used as the receiver parameter in a message call, when the method needs the services of another method in the same class.

dbobj

This message is used to retrieve a database object from the repository. This is a message to the run-time system object. This message requires two parameters: one, the meta type of the database object to be retrieved, and two, the query for the object. If the query is NULL, then the first available object of the type is retrieved. The following example shows how this message can be used:

```
/* retrieves the object of type PROJECT whose TITLE is
"Proj 1" */
a = dbobj (PROJECT, TITLE="Proj 1");
```

dbrel

This message is used to retrieve relationships from the repository. Its syntax is similar to the dbobj message. The first parameter is the metatype of the relationship being retrieved, and the subsequent parameters are RSI2 objects representing database objects involved in this relationship. This message is sent to the run-time system object. The following example depicts the usage of this message:

```
/* retrieves the object of type PROJECT whose TITLE is
"Proj 1" */
a = dbobj (PROJECT, TITLE="Proj 1");
```

```
/* now retrieves the relationship of type PROJECT-ROLES
involving a in the PROJECT role */
b = dbrel (PROJECT-ROLES, a);
```

The following example shows how the two statements above can be combined into one.

```
/* retrieves the relationship of type PROJECT-ROLES
given the object in the PROJECT role */
b = dbrel (PROJECT-ROLES, dbobj (PROJECT,
                                 TITLE="Proj 1"));
```

dbprop

This database query message retrieves a property from the repository. The parameters, as before, are the property meta type, and the object, relationship, or role whose property is needed. This is also a message sent to the run-time system object. An example:

```
/* retrieves the property of type START-DATE for the
relationship of type PROJECT-ROLES given the object in
the PROJECT role */
b = dbprop (START-DATE, dbrel (PROJECT-ROLES,
                               dbobj (PROJECT,
                                      TITLE="Proj 1")));
```

operators

The following are the operators available in RSI2. They represent the primitive operations possible in the language. They have the usual meaning: +, -, *, /, LT, LE, GT, GE, NE, =.

Using RSI2 from the Command Line

RSI2 is designed to be used, at present, from the OS/2 command line. This section details the command to be typed in to invoke RSI2.

Running the Compiler

The compiler is run using the rsicom command on the command line of the OS/2 operating system:

rsicom progname metadb -o exename

progname - the name of the RSI2 source file

metadb - the meta database

-o - output to file exename

-b - number of buffers

-d - debug mode

-s - provide source listing

Running the Interpreter

The interpreter is run using the rsirun command on the command line of the OS/2 operating system:

```
rsirun progname metadb instdb -o filename
```

progname - the name of the RSI2 executable file

metadb - the meta database

instdb - the instance database

-o - output to file filename

-b n - number of buffers

-l - library files to be linked

APPENDIX D

LIST OF ACRONYMS USED

CASE	Computer-Aided Software Engineering
DLL	Dynamic Link Library
IDO	Id of Database Object
LBMS	Learmonth and Burchett Management Systems, Inc.
OOP	Object-Oriented Programming
OPRR	Objects, Properties, Roles, Relationships
RSI	Report Specification Interface
UID	Unique Id
YACC	Yet Another Compiler Compiler

VITA 2

Manohar S.V. Rao

Candidate for the Degree of

Master of Science

Thesis: AN OBJECT-ORIENTED LANGUAGE FOR THE REPORT SPECIFICATION INTERFACE TO A CASE REPOSITORY

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangalore, India, April 24, 1967, the son of Vasudeva and Shantha Rao.

Education: Received Bachelor of Engineering Degree in Mechanical Engineering from the University of Mysore, India in 1988; completed requirements for the Master of Science degree at Oklahoma State University in May, 1992.

Professional Experience: Graduate Assistant, Department of Business Administration, Oklahoma State University, March, 1990, to October, 1990. Presently working as Software Engineer for LBMS Inc., based in Houston, Texas.